Bounding the Cost of Learned Rules: A Transformational Approach

Jihie Kim
USC/Information Sciences Institute

December 1996
ISI/RR-96-452

USC
UNIVERSITY
OF SOUTHERN
CALIFORNIA

INFORMATION SCIENCES INSTITUTE
School of Engineering / 4676 Admiralty Way, Suite 1001
Marina del Rey, California 90292-6695 / 310 822 1511

REPORT DOCUMENTATION PAGE

FORM APPROVED
OMB NO. 0704-0188

1. AGENCY USE ONLY ***(Leave blank)***

2. REPORT DATE
December 1996

3. REPORT TYPE AND DATES COVERED
Research Report

4. TITLE AND SUBTITLE

Bounding the Cost of Learned Rules: A Transformational Approach

5. FUNDING NUMBERS

ARPA/NRL/Michigan Univ.:
N00014-92-K-2015

ARPA/NRAD:
N66001-95-C-6013

6. AUTHOR(S)

Jihie Kim

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

USC INFORMATION SCIENCES INSTITUTE
4676 ADMIRALTY WAY
MARINA DEL REY, CA 90292-6695

8. PERFORMING ORGANIZATON REPORT NUMBER

ISI/RR-96-452

9. SPONSORING/MONITORING AGENCY NAMES(S) AND ADDRESS(ES)

ARPA
3701 N. Fairfax Dr.
Arlington, VA 22203

University of Michigan
Ann Arbor, MI 48109

Naval Research Lab
4555 Overlook Ave., SW
Wash., D.C. 20375

NRaD
53560 Hull St.
San Diego, CA 92152

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12A. DISTRIBUTION/AVAILABILITY STATEMENT

UNCLASSIFIED/UNLIMITED

12B. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 words)***

Efficiency is a major concern for all problem solving systems.One way of achieving efficiency is the application of learning techniques to speed up problem solving. However, many speed-up learning systems suffer from the utility problem: the cost of using the learned knowledge often overwhelms its benefit, so that the problem solving time after learning is greater than the problem solving time before learning. Assuring that learned knowledge will in fact speed up system performance has been a focus of research in explanation-based learning (EBL).

One way of finding a solution which can guarantee the cost boundedness is to analyze all the sources of cost increase in the learning process and then eliminate these sources. This thesis demonstrates how the cost increase of a learned rule in an EBL system can be analyzed by characterizing the learning process as a sequence of transformations. The learning process is decomposed into a sequence of transformations that go from a problem solving episode, through a sequence of intermediate problem solving/rule hybrids, to a learned rule.

Such an analysis has been performed on Soar (a problem solving system with a variant of EBL). By decomposing the learning process into a sequence of transformations, and analyzing these transformations, the causes which can make the output rule expensive have been identified. This analysis has also pointed the way toward a set of modifications of the transformational sequence that could potentially eliminate these causes.

These modifications have been applied to Soar, and the original sequence of transformations has been converted into a new sequence of transformations. The experimental results, at least for the domains investigated, indicate that the time after learning is consistently less than the time before learning with the new learning algorithm.

14. SUBJECT TERMS

chunking, explanation-based learning, speed-up learning, Soar, transformation, utility problem

15. NUMBER OF PAGES
174

16. PRICE CODE

17. SECURITY CLASSIFICTION OF REPORT
UNCLASSIFIED

18. SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

19. SECURITY CLASSIFICATION OF ABSTRACT
UNCLASSIFIED

20. LIMITATION OF ABSTRACT
UNLIMITED

GENERAL INSTRUCTIONS FOR COMPLETING SF 298

The Report Documentation Page (RDP) is used in announcing and cataloging reoprts. It is important that this information be consistent with the rest of the report, particularly the cover and title page. Instructions for filling in each block of the form follow. It is important to stay within the lines to meet optical scanning requirements.

Block 1. Agency Use Only (Leave blank).

Block 2. Report Date. Full publication date including day, month,a nd year, if available (e.g. 1 jan 88). Must cite at least the year.

Block 3. Type of Report and Dates Covered. State whether report is interim, final, etc. If applicable, enter inclusive report dates (e.g. 10 Jun 87 - 30 Jun 88).

Block 4. Title and Subtitle. A title is taken from the part of the report that provides the most meaningful and complete information. When a report is prepared in more than one volume, repeat the primary title, add volume number, and include subtitle for the specific volume. On classified documents enter the title classification in parentheses.

Block 5. Funding Numbers. To include contract and grant numbers; may include program element numbers(s), project number(s), task number(s), and work unit number(s). Use the following labels:

C	- Contract	PR	- Project
G	- Grant	TA	- Task
PE	- Program Element	WU	- Work Unit Accession No.

Block 6. Author(s). Name(s) of person(s) responsible for writing the report, performing the research, or credited with the content of the report. If editor or compiler, this should follow the name(s).

Block 7. Performing Organization Name(s) and Address(es). Self-explanatory.

Block 8. Performing Organization Report Number. Enter the unique alphanumeric report number(s) assigned by the organization performing the repor.

Block 9. Sponsoring/Monitoring Agency Names(s) and Address(es). Self-explanatory

Block 10. Sponsoring/Monitoring Agency Report Number. (If known)

Block 11. Supplementary Notes. Enter information not included elsewhere such as: Prepared in cooperation with...; Trans. of ...; To be published in... When a report is revised, include a statement whether the new report supersedes or supplements the older report.

Block 12a. Distribution/Availability Statement. Denotes public availability or limitations. Cite any availability to the public. Enter additional limitations or special markings in all capitals (e.g. NOFORN, REL, ITAR).

DOD - See DoDD 5230.24, "Distribution Statements on Technical Documents."
DOE - See authorities.
NASA - See Handbook NHB 2200.2.
NTIS - Leave blank.

Block 12b. Distribution Code.

DOD - Leave blank.
DOE - Enter DOE distribution categories from the Standard Distribution for Unclassified Scientific and Technical Reports.
NASA - Leave blank.
NTIS - Leave blank.

Block 13. Abstract. Include a brief (Maximum 200 words) factual summary of the most significant information contained in the report.

Block 14. Subject Terms. Keywords or phrases identifying major subjects in the report.

Block 15. Number of Pages. Enter the total number of pages.

Block 16. Price Code. Enter appropriate price code (NTIS only).

Blocks 17.-19. Security Classifications. Self-explanatory. Enter U.S. Security Classification in accordance with U.S. Security Regulations (i.e., UNCLASSIFIED). If form contins classified information, stamp classification on the top and bottom of the page.

Block 20. Limitation of Abstract. This block must be completed to assign a limitation to the abstract. Enter either UL (unlimited) or SAR (same as report). An entry in this block is necessary if the abstract is to be limited. If blank, the abstract is assumed to be unlimited.

Dedication

To My Family

Acknowledgements

First of all, I wish to express my deepest gratitude to my advisor, Paul Rosenbloom, for his guidance and support. Throughout my Ph.D. program, he shaped my development as a researcher and provided me with the freedom and support on my research and the dissertation. I am grateful to Craig Knoblock and Stephen Read, my dissertation committee, and also grateful to Shankar Rajamoney and Ken Goldberg, my guidance committee, for many helpful comments and suggestions.

I would like to thank the members of the ISI Soar group, including Richard Angros, Johnny Chen, Bonghan Cho, Jon Gratch, Randy Hill, Lewis Johnson, Gal Kaminka, Soowon Lee, Ben Smith, and Milind Tambe for interesting discussion and helpful input on this research. Especially, I owe a great debt to Jon who provided me with wise answers whenever I had dubious questions. He is not only a good helper, but also a terrific friend. Milind also gave me detailed comments to fill the holes in my presentation.

I would like to thank John Laird, Jill Lehman, Bob Doorenbos, and Scott Huffman, members of the Soar group for their comments and encouragement. Also, I wish to thank Tom Mitchell, Haym Hirsh, Steve Minton, and Ray Mooney for their technical comments and advice on my work.

I found my experience in the ISI robot project extremely valuable and I am grateful to Wei-Min Shen, Jafar Adibi, Behnam Salemi, Sheila Tejada for such opportunity. I wish to thank my friends at ISI and USC, including Hans Chalupsky, Yolanda Gil, Kevin Knight, Ping Luo, Eric Melz, Jose-Luis Ambite-Molina, and Lorna Zorman for their assistance, support, and for making the time at ISI enjoyable. I also would like thank Velda Thomas for her help in editing this thesis.

I would like to thank Suk I. Yoo, my advisor at the Seoul National University, for inspiring me to step into artificial intelligence, and providing me with constant encouragement.

I am grateful to my family. My parents, parents-in-law, brother, sister, and sisters-in-law provided their constant love, encouragement, and support. Especially my parents for their love and support in every endeavor that I have ever taken upon myself, and for encouraging me to pursue my studies. Last, but by no means least, I thank my husband, Dongho, who provided love and encouragement throughout. Without his patient understanding and support, I could not have pulled through the difficult times. Having him there to share my successes made them more meaningful. I thank him, most of all, for making it all matter.

This research was supported under subcontract to the University of Southern California Information Sciences Institute from the University of Michigan, as part of contract N00014-92-K-2015 from the Advanced Systems Technology Office (ASTO) of the Advanced Research Projects Agency (ARPA) and the Naval Research Laboratory (NRL); and under contract N66001-95-C-6013 from the Advanced Systems Technology Office (ASTO) of the Advanced Research Projects Agency (ARPA) and the Naval Command and Ocean Surveillance Center, RDT&E division (NRaD).

Contents

List Of Figures

Chapter 1

Introduction

Problem solving is an activity of an agent (or a set of agents) to achieve its (their) goals. Given a goal and a problem formulation, which consists of specifications of actions and states to be considered, a problem solver's task is to find a sequence of actions that gets an agent (or a set of agents) to a goal state from the current state. (The process of finding a sequence of actions is also called *search* in general.) For example, solving a scheduling problem for a production line, or controlling tactical aircraft in distributed battlefield simulations to accomplish a mission, or simply playing an eight-puzzle game are all problem solving activities. (From now on, I will refer to the problem solving activity as the *problem solving* for brevity.)

Efficiency is a major concern for all problem solving systems. Depending on the tasks to be performed, the problem solver (the agent) has to achieve a certain level of efficiency. For example, building a chess playing agent that can challenge the world chess champion requires a way of computing effective moves in a reasonable response time. One way of achieving such efficiency is the application of learning techniques to speed up problem solving. For example, after solving a complex problem, the system can remember how it solved the problem, and generalize the experience to solve related problems more easily. However, Minton [41] has identified that the overhead of using learned knowledge often overwhelms its benefit. The problem solving time with the learned knowledge can become greater than the problem solving time without it, because the match cost of the learned knowledge against the current state is greater than the savings by the learned knowledge. This phenomenon is called the *utility problem*, and it has turned out to be pervasive in many learning systems that are intended to speed up problem solving.

One way of understanding how learned knowledge can slow down problem solving is to analyze how the learning system can produce expensive knowledge. By investigating the underlying learning algorithm and analyzing which sub-parts contribute to producing expensive-to-use knowledge, we can find the sources of expensiveness in the learning process. When the learning algorithm can be characterized as a sequence of steps, such as a sequence of transformations from the problem solving episode to the learned knowledge, this analysis can be performed by examining each step carefully. In this case, each transformation changes one intermediate product (or problem solving episode for the first transformation) into another (or learned knowledge for the last transformation). By computing and comparing the cost of the intermediate products, the changes in cost as a result of a transformation can be measured and isolated. Whenever a transformation increases the cost, that step can be considered as a source of expensiveness. The key element required for the analysis is a tool for computing the cost of the intermediate products.

In addition to identifying which transformations lead to cost increases, and how they lead to such increases, the analysis may also point the way toward modifications of the transformational sequence that could potentially eliminate these cost increases. Once these modifications are identified, they should be carefully performed, so that the interactions across the modified subprocesses do not create another source of expensiveness. After the modifications eliminate the sources (without introducing extra sources of expensiveness), we can guarantee that the learning system will not slow down the problem solver.

This dissertation demonstrates such an analysis in the context of Soar (a problem solving system with a variant of explanation-based learning). The learning process is characterized as a sequence of transformations where the cost of intermediate products can be measured. By analyzing cost changes through the transformations, the causes which can make the output rule expensive are identified. Based on the causes and the proposed modifications, a new learning algorithm is developed.

1.1 The Goal of the Thesis

This section describes related issues and specifies the goal of the thesis. The details of the related work are given in Chapter 8.

Many speed-up learning systems acquire new knowledge in the form of *search-control rules.* (Other types of knowledge will be discussed in Chapter 8.) Search-control rules guide problem solving by indicating which paths are more promising than others, or which paths lead to failures. This guidance prunes the search branches and can reduce problem solving time. The utility problem for these systems is mainly concerned with the cost of using the learned rules. The research on this problem has raised two key issues. The first issue is the *expensive-rule problem* [61], in which individual learned rules are so expensive to match that the system suffers a slow down from learning [41, 61, 12, 59, 60]. The second issue is the *average-growth effect* [9], where the interactions across the rules slow down the system, even if none of the rules individually are all that expensive. Recent work on the average growth effect has developed a set of optimizations of the match algorithm to reduce slowdown due to learning a large number of rules [9]. Although the optimizations cannot completely eliminate potential causes of slowdown, the work has shown that in some tasks, it is possible to learn over one million rules while still allowing their efficient use. Our research focuses on the expensive-chunk problem. The solutions to both issues, expensive-chunk problem and the average-growth effect, eventually must be combined, but that is a topic for future work.

There are various approaches for speed-up learning, including those using inductive techniques [22, 33, 55, 18]. Our work focuses on EBL, the most widely used speed-up technique [41, 7, 14, 27, 23], rather than on these techniques. Once we solve the utility problem in EBL, the results may help guide similar analyses of other speed-up learning techniques.

One class of approaches to the expensive-rule problem is *discriminatory learning* [41, 19, 16, 39]. After evaluating the utility of learned knowledge, the system keeps only the useful ones by comparing the cost of using the new knowledge and its benefit. However, the utility evaluation of the candidate rules may become a factor of the utility problem if it requires extensive computations [17]. Also, the system may waste a lot of energy in learning and evaluating the knowledge when a large part of the learned rules turns out to be useless. If most of the learned rules are useful in the future (though not necessarily now), throwing away them loses the full benefit of the learned knowledge. This research excludes this class of approaches, and is aimed at learning cheap rules in the beginning.

One way of solving the expensive-rule problem is to ensure that the cost of using the learned rule is bounded by the cost of problem solving without the learned rule. A learned rule is called "expensive" when its match cost is greater than the cost of the problem solving from which it is learned. An optimization which can reduce the match cost of a learned rule does not necessarily solve the expensive-rule problem unless it guarantees such boundedness. For example, Prieditis et al. [50, 41, 59, 58, 12] have investigated how to produce cheaper rules. (The details of these approaches are described in Chapter 8.) However, none of these approaches can guarantee that the cost of using the learned rules will always be bounded by the cost of the problem solving episode from which they are learned. Thus the goal of solving the expensive-rule problem is split from simply reducing the match cost of the learned rules. However, with this definition of expensiveness, if the original problem solving has required exponential search, then the run time after learning could be exponential, though still not worse than the run time before learning. Thus the goal of removing expensive rules has been split again from the goal of guaranteeing a bound on the match. A solution that achieves the former goal but not the latter one is called a *relative solution.* An *absolute solution* is defined as one that provides a guaranteed bound on the match of the learned rules regardless of the original problem-solving cost. The relative solution, however, is still important to be examined. While getting a bound on match, the absolute solutions [61, 65] which have been investigated so far, have several drawbacks. Not only do they reduce the expressibility of the system, but they also reduce the generality of the rules. They sometimes require a large number of rules for the same knowledge which can be expressed by a single rule [61]. This research focuses on the relative solution.

The goal of this research is to provide a relative solution without restricting the expressiveness of the learned knowledge. In other words, we want to *make sure that the cost of using learned rules is no more than the cost of problem solving.* This will make the cost of using the learned rules always be bounded by the cost of the problem solving episode from which they are learned. One way of providing such boundedness is to: (1) *find the complete set of sources that can make learned rules expensive*, and then (2) *modify the learning process to avoid these sources.* To find the set of sources of expensiveness, this research introduces a novel way of analyzing the learning process called the *transformational analysis.* The essence of the approach is *to decompose the learning process into a sequence of transformations in which the cost of intermediate* *products can be computed and compared.* The EBL algorithm that goes from a problem solving episode to a learned rule can be decomposed into a sequence of transformations that change one intermediate problem solving/rule hybrid (called a *pseudo-chunk*) into another. (*Chunk* here means any learned rule. This is a generalization of the term used in the Soar [35, 54] system.) For example, filtering out unnecessary rules which did not participate in the problem solving can be the first transformation in the learning process. As the sequence progresses, the pseudo-chunks become more like rules and less like the problem solving. (This is an approximate description of the transformational analysis. The details describing this process will be provided in Chapter 2.)

The key difference between the transformational analysis and the standard analysis of EBL is that the cost of intermediate products can be computed in the transformational analysis. In the standard analysis, learning is a process consisting of a sequence of sub-processes which create *non-executable* intermediate products. Here, the intermediate products (pseudo-chunks) are *executable* in that they can be matched and fired (given an appropriate interpreter) and thus independently create the same effect as the problem solving episode. By computing and comparing the match cost of each pseudo-chunk, the cost changes throughout the learning process can be measured and isolated within the steps in which the transformations occur. Once the sources of extra cost are found, by avoiding those sources, the cost of the learned rule can be bounded by the cost of the problem solving.

The following section provides an example of transformational analysis. It analyzes one transformation in the EBL algorithm, and shows how we can find a source of expensiveness through the analysis, and how a solution can be provided to avoid the source.

1.2 An Example of the Transformational Analysis

As described above, a problem can be solved by finding a sequence of operators (a path) leading to the goal state from the initial state. When the problem solver employs *search-control rules*, rules that determine which operators are selected for which states, the actual path depends on these rules. Figure 1.1 shows an abstract view of a sequence of operator applications. The gray arrows denote the search-control rules which affect each decision.

When a new rule is acquired from a trace of the problem solving, the control rules are often removed in the learning process. That is, the instantiations of the search-control rules

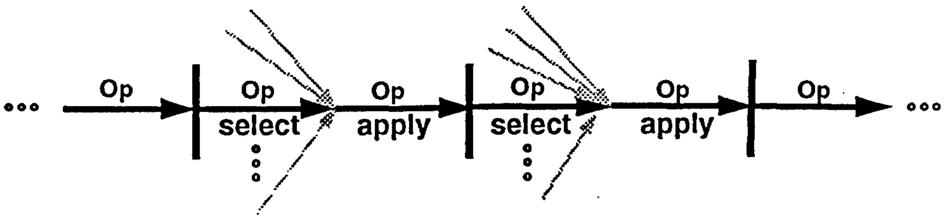
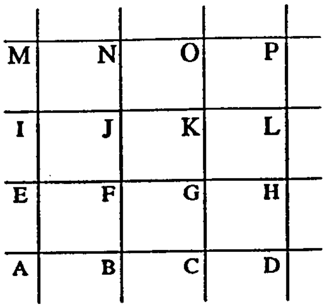


Figure 1.1: A sequence of decisions affected by search-control rules.

are not included in the explanation. For instance, PRODIGY/EBL [41] and Soar [37, 54] — two problem solvers that learn search-control rules by a variant of EBL — ignore a large part of the existing search-control rules in learning, in order to increase the generality of the learned rules.[1] (The details of how this transformation increases the generality are in Chapter 3.) The most critical consequence of this transformation (removal of search control) is that the learned rule (and the pseudo-chunks created between the transformation and the learned rule) are not constrained by the path actually taken in the problem space. Thus, they can perform an exponential amount of search even when the original search was highly directed (by the control rules).

Consider an example from the Grid task — a task known to suffer from the expensive-rule problem [61] — as shown in Figure 1.2. Each problem in the Grid task is to find a path between two points in a two dimensional grid. The given problem is to go from point F to point P, a path of length four. In Figure 1.2-(a), each connection between two points is represented by a tuple that contains three items: object identifier, attribute (^ indicates attribute name), and value. For example, the connection between point F and point J is represented by the tuple (F ^next J). Figure 1.2-(b) shows a rule that proposes a candidate operator. The symbols enclosed in angle brackets are variables. The rule *operator-goto* says that if the location of the current state is point <loc1> and <loc1> is connected to another point <loc2>, then a new operator can be proposed which moves the current location from <loc1> to <loc2>. Because F is connected to four adjacent points, four operators can be suggested by the rule. With suitable control knowledge, the system can solve the problem of finding a path from point F to point P — for example, F, G, H, L, and

[1] In Prodigy, selection and rejection rules are included in the explanation, but preference rules are not [43]. Likewise, Soar currently also includes require and prohibit preferences, but not desirability preferences. The details of the preference semantics are explained in Chapter 5.



(F ^next B)
(F ^next E)
(F ^next J)
(F ^next G)
(G ^next F)
.
.
.

(a) Connections among the points

```
(sp operator-goto
    (goal <g> ^problem-space <p>
              ^state <s>)
    (<p> ^name grid-path)
    (<s> ^at <loc1>)
    (<loc1> ^next <loc2>)
-->
    (<o> ^name goto-loc
         ^from <loc1> ^to <loc2>)
    (<g> ^operator <o>))
```

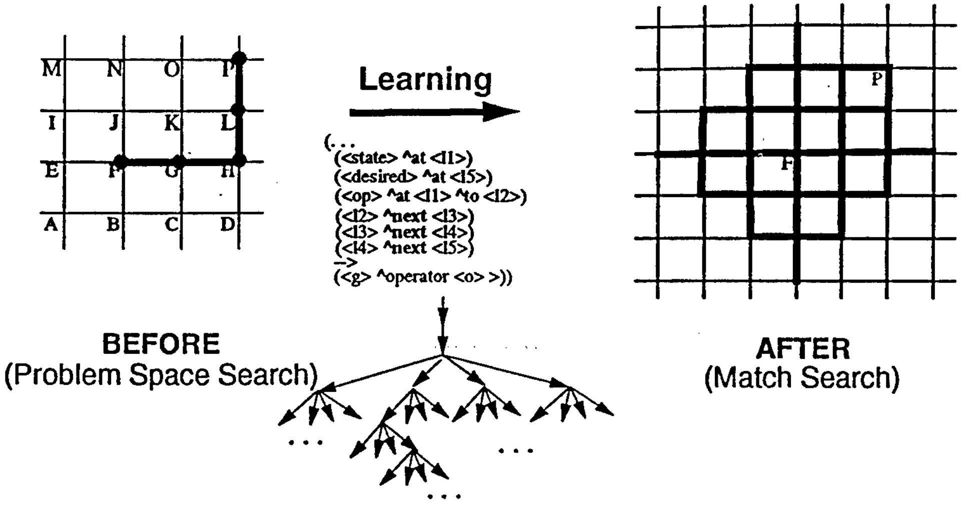
(b) Operator proposal rule

Figure 1.2: Grid task.

P — in time that is linear in the length of the path. However, the rule learned from this search may be so general that, when it matches, it searches over all paths of length four instead of just a single path.

Figure 1.3 shows the relationship between the search upon which the learning is based and the search performed, during the match, by the rule learned from this search. The rule in Figure 1.3 says that if you are at location <l1> and want to get to location <l5>, and there is an operator that takes you from <l1> to <l2>, and there is a connected path from <l2> to <l5> (via two intermediate points, <l3> and <l4>), then the operator is the best choice. This rule is quite general, as it can solve any problem that has a solution of length four and find all such paths, which is a key difference from original problem solving with search control. This generality, however, is only obtained at an enormous cost. That is, the cost is exponential in the length of the path (as shown as a set of arrows in the figure). Although, using this learned rule, the system can solve the same problem within a single rule firing instead of requiring multiple rule-firing cycles, the run time may become longer because of this exponential match search.

The above analysis of a transformation (removal of search-control knowledge) reveals one important source of cost increase in the learning: removal of search-control can increase the cost. One way of avoiding this problem is to incorporate the search-control

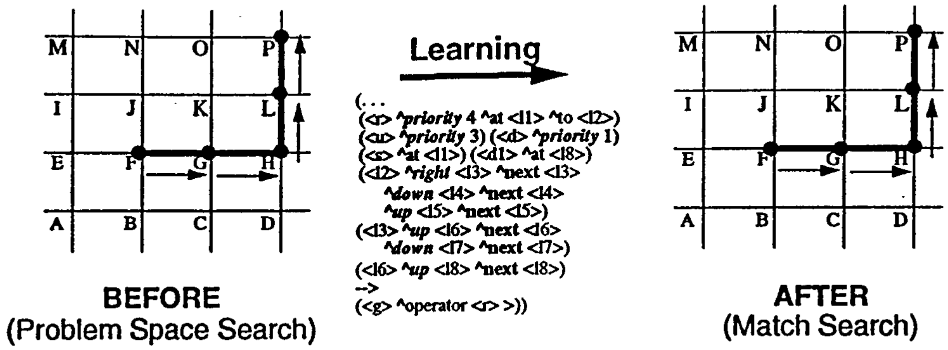


```
(...
 (<state> ^at <l1>)
 (<desired> ^at <l5>)
 (<op> ^at <l1> ^to <l2>)
 (<l2> ^next <l3>)
 (<l3> ^next <l4>)
 (<l4> ^next <l5>)
 -->
 (<g> ^operator <o> >))
```

Figure 1.3: The difference between the search during problem solving and the search during the match of the learned rule.

instead of removing it. By *incorporating the traces of control rules utilized in the problem-space search*, the match process for learned rules (and the pseudo-chunks created between the transformation and the learned rules) becomes focused on just the path that was actually followed (as shown in Figure 1.4), thus ensuring that the match search for the learned rule is bounded in complexity by the problem-space search from which it was learned. The rule in Figure 1.4 is what would be learned by this solution. This rule corresponds to the rule in Figure 1.3. The additional conditions in the rule, though they look unusual, reflect the aspects of the problem tested by the search control that was part of the problem solving. These conditions constrain the search in the match to the search in the problem solving. (The details of how to match these conditions will be given in Chapter 5.) This can specialize the learned rule, but in return it enables the rule's cost to remain bounded by the cost of the original problem solving.

This change (from removal of search control to incorporation of search control) may require modifications of the subsequent transformations. To be able to maintain the search control in the transformations, without introducing another source of cost increase, the subsequent transformations should be properly adjusted. It turned out that the above modification of the transformation requires significant changes of the subsequent transformations. The details will be discussed in Chapter 5.



```
(...
 (<r> ^priority 4 ^at <l1> ^to <l2>)
 (<u> ^priority 3) (<d> ^priority 1)
 (<s> ^at <l1>) (<d1> ^at <l8>)
 (<l2> ^right <l3> ^next <l3>
     ^down <l4> ^next <l4>
     ^up <l5> ^next <l5>)
(<l3> ^up <l6> ^next <l6>
     ^down <l7> ^next <l7>)
(<l6> ^up <l8> ^next <l8>)
-->
(<g> ^operator <r> >))
```

Figure 1.4: Searches that would be performed by including search control in learning.

1.3 Overview of the Approach

To reveal all sources of additional cost, a complete analysis of the whole sequence of transformations is required, as was done for one transformation above. We have performed such a transformational analysis of learning in Soar. The sequence of transformations in the learning process has been mapped into a sequence of transformations where the cost of intermediate products can be measured. By analyzing these transformations, two additional sources that can make the output chunk expensive have been identified. First, *losing the efficiencies (such as sharing) stemming from the graph structure of the problem solving can increase the cost* [30]. During problem solving, the rules that fire tend to form a directed acyclic graph structure in which the early rules provide information upon which the firing of later rules depends. This graph structure is called the *problem-solving structure.* The problem-solving structure is reflected in EBL most obviously in the structure of the explanation (and the more general explanation structure). However, if this graph structure is then flattened into a linear sequence of conditions (via a transformation) for use in matching the rule that is learned — as must be done in creating Ops-like rules or Prolog clauses — the efficiencies stemming from the graph structure are lost, and the cost after the transformation can be greater than the cost before it. If instead, the learning mechanism is made sensitive to such efficiencies — i.e., by reflecting the graph structure in the match of the learned rule — this source of expensiveness can be avoided. This requires modifications of the match algorithm to be able to support graph structured instantiations, and adjustments of the subsequent transformations (should they exist).

The other source of expensiveness is in *disrupting the optimizations (such as the removal of duplicates) based on equivalent information*. For example, in Ops-like languages, working memory is a set, and does not allow duplicate elements. Whenever rule firings create duplicates, they are merged into one element. When multiple rule firings create the same working memory elements, only one of them is saved in the working memory and used in the future matches. If this optimization is ignored in learning new rules (as is the case in most EBL systems), time after learning can be greater than time before learning because of the extra match effort for the duplicate elements. This problem can be solved by introducing an equivalent optimization function in the learned rules. (The details are described in Chapter 5.) Also, the subsequent transformations should be modified as in the above cases.

The proposed solutions for each identified source of cost increase should be combined to produce a unified solution. However, unifying each proposed solution for each identified source of cost increase is not simply pipelining the set of solutions. For example, the solution for the first problem (incorporating search control) introduces additional rules in the explanation structure, and the solution for the second problem (introducing graph structure in the learned rule based on the explanation structure structure) should efficiently capture this additional part. That is, the system should develop a way of embodying the search-control knowledge in the graph structure.

Overall, the dual process of first finding the sources of cost increase through a transformational analysis, and then modifying the learning process based on the analysis, is called the *transformational approach*. This research investigates the transformational approach in the context of Soar. Soar is an architecture that combines general problem solving abilities with a learning mechanism called *chunking* [36]. Chunking is a variant of EBL [53], and also suffers from the expensive-rule problem [61]. The transformational analysis is presented in terms of chunking in Soar. Also, to be able to more easily generalize the resulting analysis to other EBL systems, we have implemented a general EBL algorithm in Soar (called *Soar/EBL*) [31] and analyzed its performance. The mapping between the two sequences of transformations has revealed that Soar/EBL yields the same sources of expensiveness as chunking.

Although the transformational approach is presented in terms of learning in Soar, the above dual process can be applied to any learning algorithm (including other EBL implementations) whenever it can be characterized as a sequence of transformations.

Also, we conjecture that the transformational analysis can be used as a tool for detecting changes in correctness as well as changes in cost. By employing a tool for computing the level of correctness before and after the transformations, the changes in correctness as a result of a transformation can be measured and isolated.

1.4 Contributions

The primary contributions of this thesis include the following:

1. *Performing a transformational analysis of learning process:* A novel way of analyzing learning processes is presented. The learning process is decomposed into a set of transformations where the cost of intermediate products can be computed. The analysis of each step (in terms of cost change through the transformation) is used as a tool for pointing out where extra cost is being added. Once the set of sources are found, the learning process can be modified to avoid the sources. This transformational analysis is also important for understanding other characteristics of the learning system, including the correctness changes through the learning process.

2. *Finding sources of expensiveness:* Through the transformational analysis, we have found three sources of expensiveness: (1) removing search-control knowledge through learning, (2) losing efficiencies (such as sharing) stemming from the problem-solving structures, and (3) disrupting the optimizations (such as removal of duplicates) based on equivalent information.

3. *Identifying solutions to sources of expensivenss:* Solutions are presented for each source of expensiveness.

   (a) *Removing search control ⇒ incorporate search control in learning*. By incorporating search control in the explanation structure, the match process for the learned rule can focus on the path that was actually followed.

   (b) *Losing efficiencies stemming from the problem-solving structures ⇒ keep the problem-solving structure*. By keeping the graph structure employed in the problem solving, the efficiencies can be reinstated.

(c) *Disrupting the optimizations based on equivalent knowledge ⇒ preprocess knowledge before it is used.* By preprocessing the knowledge either by grouping the equivalent knowledge or by selecting one of them as a representative, an equivalent optimization can be achieved.

4. *Finding a unified solution to the set of sources:* The proposed solutions are combined to produce a unified solution. This requires an efficient synthesis of the solutions, so that one does not hinder another. This unified solution guarantees that it avoids the sources of expensiveness each solution is trying to avoid.

5. *Developing extensions to the match algorithm:* To interpret the intermediate products and evaluate the costs of the products, extensions to the match algorithm are required. Also, the unified solution demands significant change of the match algorithm. These different sets of extensions have been implemented.

6. *Evaluating the unified solution via experimental results:* The unified solution has been implemented, and the results from the new learning system show that the cost of using the learned rule is bounded by the cost of the problem solving.

7. *Mapping EBL onto Soar:* The transformational analysis performed for chunking has been mapped onto the sequence of transformations for EBL. The mapping contrasts the differences in cost and correctness between the two learning systems.

1.5 Organization of the Thesis

The body of this thesis consists of eight chapters. Chapter 2 describes the core idea of the transformational approach. It characterizes the standard EBL algorithm as a sequence of transformations as described above. The chunking algorithm is also characterized as a sequence of transformations in the same way. Chapter 3 presents a transformational analysis performed on chunking in Soar. Each transformation is examined in detail, and the cost changes through the transformations are analyzed. The key result of these analyses is the identification of new sources of expensiveness and presentation of solutions for each source. Chapter 4 maps the analysis performed for chunking to Soar/EBL. The analysis demonstrates that EBL, as implemented for Soar, yields the same sources of expensiveness and overgenerality as does chunking. Chapter 5 presents a unified solution based on the set of individual solutions. The details of each solution are described, and the modifications required to combine all solutions are presented. Chapter 6 presents detailed results from the implementation of the unified solution discussed in Chapter 5. Chapter 7 and Chapter 8 discuss related work and the conclusion, respectively.

Chapter 2

Core Idea of Transformational Analysis

This chapter describes how the EBL utility problem can be investigated via a transformational analysis of the learning system. First, the standard description of EBL as a sequence of transformations is reviewed. Then the intermediate products (and the input and the output) of the sequence are mapped into executable structures (called the *pseudo-chunks*) so that the cost of intermediate products can be measured and compared. Finally, we characterize chunking as a sequence of transformations, with their intermediate pseudo-chunks, in the same way.

2.1 EBL as a Sequence of Transformations

2.1.1 An overview of EBL

Given:

(1) **Goal concept** : A definition of the concept to be learned.

(2) **Training example** : A specific example of the goal concept.

(3) **Domain theory** : A set of rules and facts to be used for proving that the training example is an instance of the goal concept.

(4) **Operationality criterion** : A specification of how the concept should be expressed.

Determine:

A generalization of the training example that is a sufficient concept description for the goal concept and that satisfies the operationality criterion.

Figure 2.1: EBL specification (adapted from [45]).

(1) **Goal Concept:** (<x> ^isa cup)

(2) **Training Example** (WMEs):

W1: (O1 ^has-part C1)
W2: (C1 ^isa concavity)
W3: (C1 ^is upward-pointing)
W4: (O1 ^has-part B1)
W5: (B1 ^isa bottom)
W6: (B1 ^is flat)
W7: (O1 ^is light)
W8: (O1 ^has-part H1)
W9: (H1 ^isa handle)
W10: (B1 ^owner Bill)

(3) **Domain Theory** (rules):

```
R1)                          R2)                      R3)                      R4)
(<x> ^has-part <y>)          (<x> ^has-part <y>)      (<x> ^is light)          (<x> ^is open-vessel)
(<y> ^isa concavity)         (<y> ^isa bottom)        (<x> ^has-part <y>)      (<x> ^is stable)
(<y> ^is upward-pointing)    (<y> ^is flat)           (<y> ^isa handle)        (<x> ^is liftable)
-->                          -->                      -->                      -->
(<x> ^is open-vessel)        (<x> ^is stable)         (<x> ^is liftable)       (<x> ^isa cup)
```

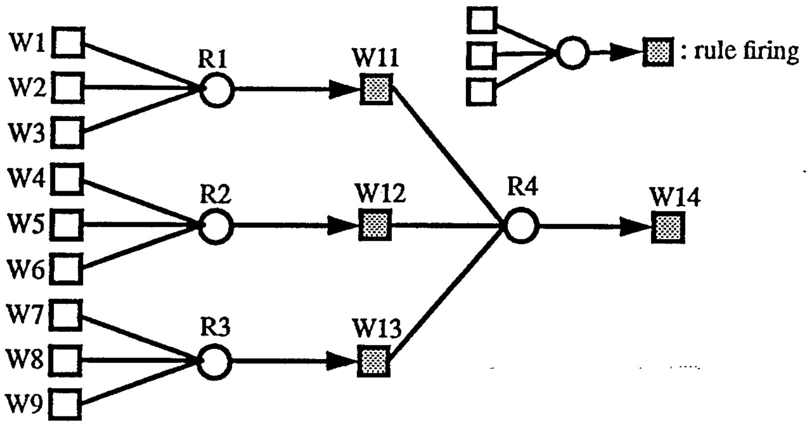
(4) **Operationality Criterion:**

Concept definition must be expressed in terms of structural features used in describing examples (e.g., light, handle, flat, etc.).

Figure 2.2: EBL input for learning the cup concept (adapted from [45]).

Explanation-based learning (EBL) is a learning paradigm for acquiring a concept from the combination of a single example and underlying domain knowledge. Figure 2.1 shows the input and the output of EBL as specified in [45]. Given the four informational components (the *goal concept*, the *training example*, the *domain theory*, and the *operationality criterion*), EBL produces a rule which describes a sufficient condition for the goal concept. An example EBL input is shown in Figure 2.2. The example is an adaptation to the Soar syntax of the cup domain, a typical illustrative EBL task, described in [45]. EBL has to learn a structural definition of a cup. There are four rules representing the domain theory and ten working memory elements (or WMEs) representing the training example. The operationality criterion specifies that the concept should be expressed in terms of the features used in the example.

Given the input, the system constructs an *explanation* (also called a *proof tree*) of how the training example is an instance of the goal concept. The explanation built from the cup domain is shown in Figure 2.3. This structure shows how the object O1 is an instance of the cup class. Each circle with its attached lines represents an instantiation of a rule in the domain theory. The lines attached to the left-hand side of a circle represent the instantiation of the conditions, and the right-hand side arrows represent the instantiations of the actions. A white square represents a fact in the domain theory. A gray square linked



Created WMEs during proof:

W11: (O1 ^is open-vessel)
W12: (O1 ^is stable)
W13: (O1 ^is liftable)
W14: (O1 ^isa cup)

Figure 2.3: An explanation for the cup domain.

to the right side of a rule instantiation is a fact (WME) produced by firing the rule. The rightmost square is an instantiation of the goal concept. The leftmost squares represent the part of the training example which participated in the explanation. In the example, among the given WMEs, W1, . . . ,W9 participate in the explanation, and W10 is excluded from it.

An *explanation structure* is built from an explanation by replacing the rule instantiations with the rules. The variable names are replaced with unique names so that there are no common variables across the rules. An explanation structure built from the explanation in Figure 2.3 is shown in Figure 2.4. The instantiation of R1 is replaced by a copy of R1 (R1′), which is the same as R1 except for the variable names. Its variable names are unique compared to the variable names of the other rules. Other rule instantiations are also replaced by rules in the same fashion.

Given the explanation structure, a variable unification process (called *regression*) is applied to it. For example, the regression algorithm in EGGS builds a substitution list based on action and condition pairs which are juxtaposed in the explanation structure [46]. In Figure 2.4, the action of R1′ is unified with the first condition of R4′, and this unification creates a substitution in which the variable <x2> is replaced by the variable <x4>. After collecting a set of variable substitutions through the unifications, EBL

```
R1')                                R2')
(<x1> ^has-part <y1>)               (<x2> ^has-part <y2>)
(<y1> ^isa concavity)               (<y2> ^isa bottom)
(<y1> ^is upward-pointing)          (<y2> ^is flat)
-->                                 -->
(<x1> ^is open-vessel)              (<x2> ^is stable)

R3')                                R4'),
(<x3> ^is light)                    (<x4> ^is open-vessel)
(<x3> ^has-part <y3>)               (<x4> ^is stable)
(<y3> ^isa handle)                  (<x4> ^is liftable)
-->                                 -->
(<x3> ^is liftable)                 (<x4> ^isa cup)
```

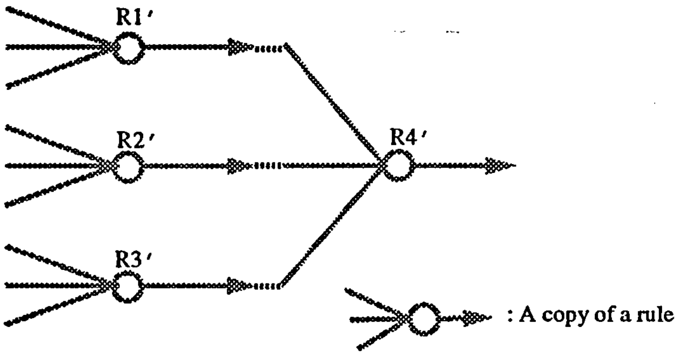


Figure 2.4: An explanation structure for the cup domain.

applies the collected substitutions to the variables in the explanation structure, and this process creates a regressed structure as shown in Figure 2.5.

After the regression, a new definition, i.e., a sufficient condition for the goal concept, is generated from the leaves of the regressed structure. The new rule is shown in Figure 2.6. Given the definition of the operationality criterion in Figure 2.2 — the concept should be expressed in terms of the structural features used in describing examples — the new definition complies with the operationality criterion.

When EBL is employed in a problem solving system (such as Soar), the rule traces in a problem solving episode can provide the explanation of why an example is an instance of a goal concept. By following the above algorithm, a new rule can be generated by EBL and be added to the domain theory of the system. Given a similar problem, firing the learned rule can produce the same effect as the rules in the original domain theory.

Note that the goal concept in the above example is a task concept (a cup concept). When the goal concept is a meta control decision (i.e., success, failure, or preferences among multiple operators), EBL creates a search-control rule which describes a sufficient condition for the particular control decision. For example, Soar and PRODIGY employ a

```
R1")
(<x4> ^has-part <y1>)
(<y1> ^isa concavity)
(<y1> ^is upward-pointing)
-->
(<x4> ^is open-vessel)
```

```
R2")
(<x4> ^has-part <y2>)
(<y2> ^isa bottom)
(<y2> ^is flat)
-->
(<x4> ^is stable)
```

```
R3")
(<x4> ^is light)
(<x4> ^has-part <y3>)
(<y3> ^isa handle)
-->
(<x4> ^is liftable)
```

```
R4")
(<x4> ^is open-vessel)
(<x4> ^is stable)
(<x4> ^is liftable)
-->
(<x4> ^isa cup)
```

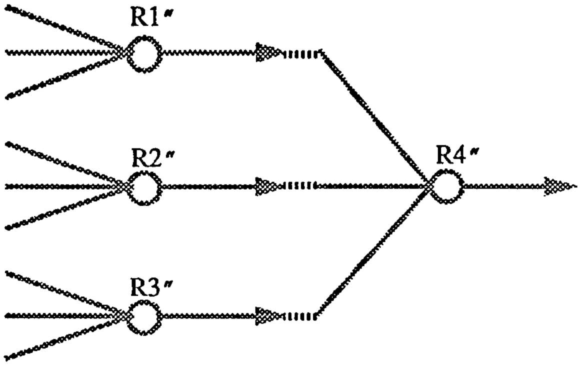


Figure 2.5: The regressed structure for the cup domain.

```
Newly-learned-rule)
(<x4> ^has-part <y1>)
(<y1> ^isa concativity)
(<y1> ^is upward-pointing)
(<x4> ^has-part <y2>)
(<y2> ^isa bottom)
(<y2> ^is flat)
(<x4> ^is light)
(<x4> ^has-part <y3>)
(<y3> ^isa handle)
-->
(<x4> ^isa cup)
```

Figure 2.6: Learned rule.

variety of goal concepts, so that they can explain why the choices made during the problem solving episode were appropriate, or were not appropriate. The new search-control rule can prune the search branches in future problem solving activities.

2.1.2 Transformational analysis of EBL

The above specification describes EBL as a sequence of transformations, where the input is the problem solving episode and the output is the chunk (the learned rule), as shown in Figure 2.7-(a). Each transformation produces a non-executable intermediate structure (e.g., explanation, or explanation structure) which is given to the next transformation. In this sequence, it is difficult to compare the costs of the intermediate products, and also difficult to analyze the effect of each transformation.

One way of analyzing the cost changes through the transformations is to map the non-executable intermediate structures into executable ones (pseudo-chunks) by providing appropriate interpreters. As shown in Figure 2.7, the original sequence of intermediate structures in EBL can be mapped into a new sequence of pseudo-chunks with their interpreters. In this new sequence, each pseudo-chunk can be matched and fired by an interpreter, and can produce the same result as the original problem solving episode. For instance, a problem solving episode can be mapped into a domain theory where its interpreter is the rule matcher and the rule firer. By computing the total cost of firing multiple rules in the domain theory, given the initial facts, we can compute the cost of the problem solving. Also, the learned rule can be simply mapped into the same learned rule with the rule matcher as its interpreter. By computing the match cost of the learned rule, we can compute the cost of interpreting the learned rule. Once we compute the cost of firing each pseudo-chunk (i.e., cost of interpreting the pseudo-chunk), we can analyze the cost changes through the transformations. The key factor here is that each pseudo-chunk should have the same effective measure of cost, so that the costs of firing different pseudo-chunks are comparable.

Although this new sequence is different from the standard sequence of transformations of EBL, both sequences are consistent in that each transformation in one sequence has a corresponding transformation in the other. For example, filtering out unnecessary rule firings in the problem solving episode is the first transformation, and it also transforms the domain theory into the next pseudo-chunk in which unnecessary rules are discarded.

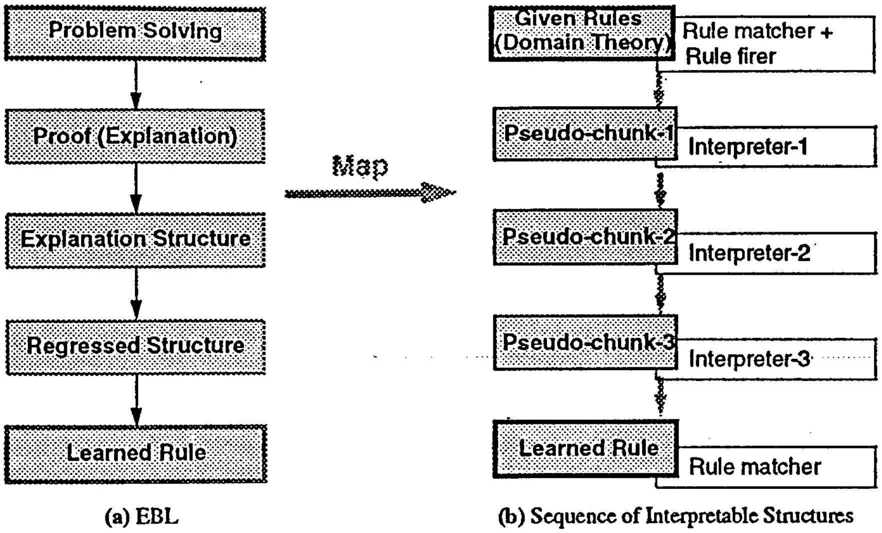


Figure 2.7: Mapping the sequence of non-executable structures into a sequence of executable structures (pseudo-chunks).

Whenever there are multiple rule firings in a pseudo-chunk, the cost of interpreting (matching) the pseudo-chunk is the total cost of matching and firing all the participating rules. For example, as described above, the cost of interpreting the domain theory is the total cost of matching and firing rules in the problem solving. Actually, the cost of firing a rule is the cost of executing actions and creating new WMEs based on the action execution. However, we will not explicitly focus on this cost for two reasons. First, the key bottleneck in rule firing is traditionally the match phase. Second, these action executions (except for the action that created an instance of the goal concept) and the WME creations drop out during learning, and do not affect the cost of the chunk; so this aspect is already guaranteed to be bounded.

2.2 Chunking as a Sequence of Transformations

Chunking is a variant of EBL, and it also can be characterized as a sequence of pseudo-chunks given appropriate interpreters (as described above for EBL), and cost changes from each transformation can be analyzed.

The cost of firing a rule is computed by the match cost of the rule; thus the cost significantly depends on the match algorithm employed in the problem solving system.

As illustrated in [11], the match algorithms employed in speed-up learning can greatly affect the utility of the learned knowledge. For example, good matchers can help avoid a part of the utility problem, and bad matchers can significantly contribute to the problem. Therefore, it is important to understand the underlying match algorithm for the utility analysis. Soar employs Rete [15] as its match algorithm. Rete is one of the most efficient rule-match algorithms presently known.

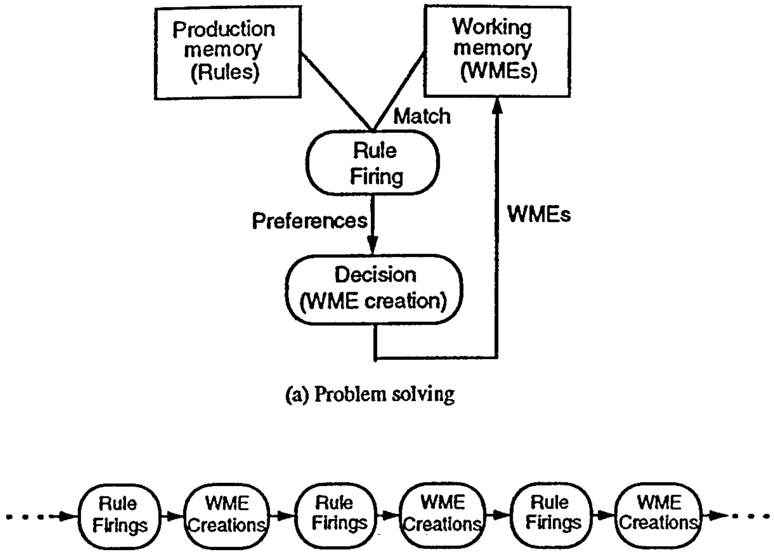
The two subsections below review chunking and the Rete algorithm. The last subsection maps chunking into a sequence of pseudo-chunks, based on the first two subsections.

2.2.1 An overview of Chunking

In Soar, productions comprise the domain theory for EBL. Each production consists of a set of conditions and a set of actions. Conditions test working memory for the presence or absence of patterns of tuples, where each tuple consists of an object identifier, an attribute, and a value. Actions create *preferences* (stored in the *preference memory*), each of which specifies the relative or absolute worth of a value for an attribute of a given object. Productions in Soar propose changes to working memory through these preferences, and do not actually make the changes themselves. Changes to working memory are made based on a synthesis of the preferences (by a fixed *decision procedure*). The cycle of production firing, creation of preferences, and creation of working memory elements (WMEs) underlie the problem solving [34].[1] Figure 2.8-(a) shows the problem solving cycle. A problem solving episode is a sequence of rule firings and WME creations, as shown in Figure 2.8-(b). This problem solving episode provides an input to the learning system.

When a unique decision cannot be made because of either incomplete or inconsistent preferences, the system reaches an *impasse*. It creates a *subgoal* to deal with the impasse. In the subgoal created for the impasse, Soar tries to resolve the impasse. Whenever a supergoal WME (called a *result*) is created in the subgoal, a new chunk is created. The result corresponds to an instantiated goal concept in EBL. The chunk summarizes the rule firings in the problem solving that produced the result in the subgoal.

[1] In fact, preferences not concerning the values of operators or states are processed as soon as they are created. Operators and states are decided only when there is no more rule firings in the system.



(a) Problem solving

(b) Problem solving episode

Figure 2.8: Problem solving in Soar.

To create chunks, Soar maintains instantiated traces of the rules which have fired in the subgoal. The operationality criterion in chunking is that the conditions in the chunk should be generated from the supergoal objects. By extracting the part of the trace which participated in the result creation, the system collects the supergoal (operational) elements connected to the result. This process is called *backtracing*, and the instantiated trace is called a *backtrace*; it corresponds to the proof tree (or explanation) in EBL. An example of rule traces is shown schematically in Figure 2.9. The two striped vertical bars mark the beginning and the ending of the subgoal. The WMEs to the left of the first bar exist in the supergoal (prior to the creation of the subgoal). The WMEs between the two bars are internal to the subgoal. The WME to the right of the second bar is the result of the subgoal. T1, T2, T3, T4, and T5 are traces of the rule firings. For example, T1 records a rule firing which examined WMEs A and B, and generated a preference suggesting WME L. In the example, T2, T3, T4, and T5 have participated in the result creation.

Instead of employing all of the rule traces which participated in the result creation, chunking only extracts traces from *task-definition rules* (rules that directly propose values of WMEs). *Search-control rules*, as distinguished from task-definition rules, suggest the relative worth of the proposed values. The search-control rules are missing in chunking

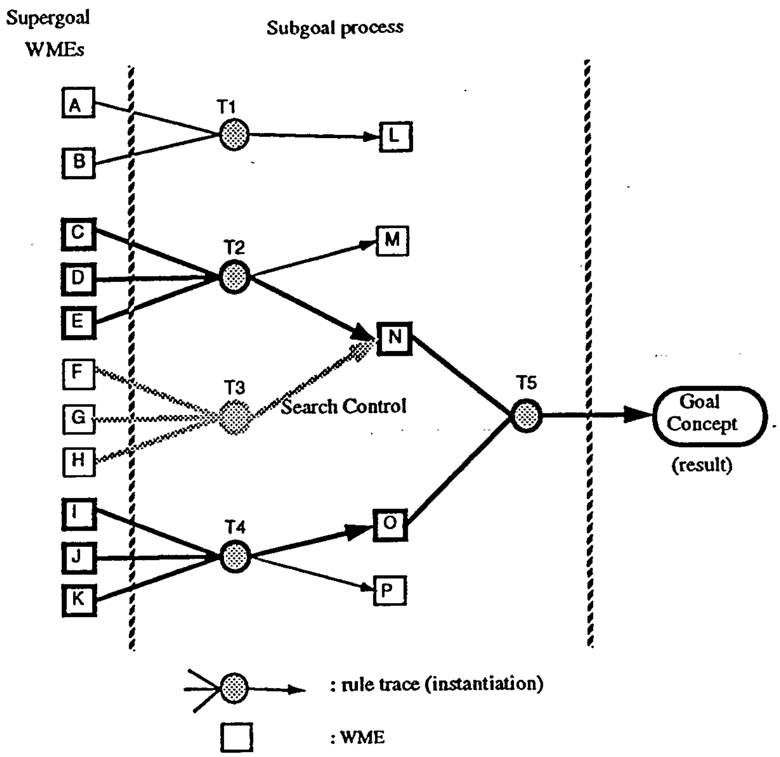


Figure 2.9: An example of the backtrace.

(and other EBL systems [43]) based on the assumption that they only affect the efficiency, not the correctness of learned rules. This omission is intended to increase the generality of the learned rules — reducing the number of conditions by leaving out search-control rules means less restriction on the test of the applicability of the rules, and thus implies increased generality. In the given example, chunking's backtrace includes T2, T4, and T5, but excludes T3 (firing of a search-control rule).

The resulting supergoal elements are variablized. The variablization step in chunking is different from the regression process in EBL in that it is performed by examining the backtrace (explanation) rather than unifying condition-action pairs in the explanation structure. All constants are left alone; they are never replaced by variables. All object identifiers in the instantiations are replaced by variables; and in particular, all occurrences of the same identifiers are replaced by the same variable. The variablized supergoal elements are reordered by a heuristic algorithm, and become the conditions of the chunk.

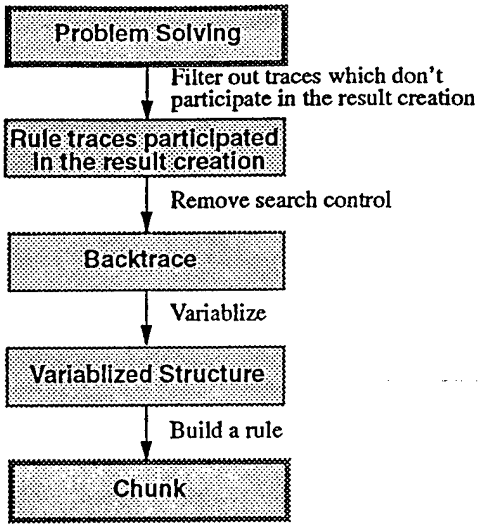

Problem Solving
Filter out traces which don't participate in the result creation
Rule traces participated in the result creation
Remove search control
Backtrace
Variablize
Variablized Structure
Build a rule
Chunk

Figure 2.10: The sequence of transformations of chunking.

The action of the chunk is the variablization of the result. This sequence of transformations of chunking is shown in Figure 2.10.

2.2.2 Rete match algorithm

In Soar, when a new rule is created, the conditions of the rule are compiled into a data flow network called a *Rete network*. The Rete algorithm's efficiency stems primarily from two key optimizations: *sharing* and *state saving*. Sharing of common conditions in a production, or across a set of productions, reduces the number of tests performed during match. State saving preserves the previous (partial) matches for use in the future. Figure 2.11 illustrates a Rete network for a rule. Rete requires a total ordering on the conditions of a rule for it to be compiled, so the rule's conditions are first ordered. For example, the conditions in Figure 2.11 are ordered (C1, C2, C3). Soar employs a heuristic ordering algorithm to improve the match performance.

The network has two parts, the alpha part and the beta part. The alpha part performs constant tests on WMEs, such as tests for **at** and **yes**. The outputs of these tests are stored in alpha memories. Each alpha memory contains the set of WMEs that pass all of the constant tests of a condition (or more than one, if it is shared). The beta part of the network contains join nodes and beta memories. There also are negative nodes, into

Rete network for one production with condition:
    C1 : (<state> ^at <loc1>)
    C2 : (<loc1> ^next <loc2>)
    C3 : (<loc2> ^goal-point yes)
when Working Memory contains
    W1 : (S1 ^at L1)
    W2 : (L1 ^next L2)
    W3 : (L1 ^next L3)
    W4 : (L2 ^goal-point yes)
    W5 : (L2 ^next L3)
    W6 : (L4 ^goal-point yes)

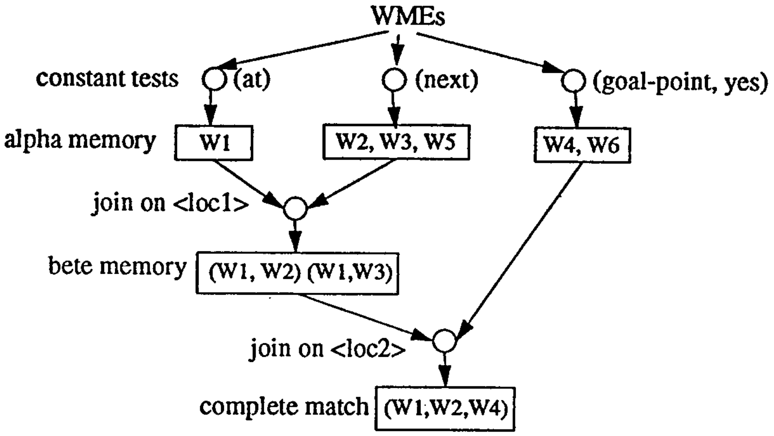


Figure 2.11: Rete network of a rule.

which negated conditions are compiled. A negative node passes a partial instantiation when there are no consistent WMEs. Join nodes perform consistency tests on variables shared between conditions, such as <loc1>, which is shared between C1 and C2. Beta memories store partial instantiations of productions, that is, instantiations of initial subsequences of conditions. The partial instantiations are called *tokens*.

We use the number of tokens as an analytic tool for measuring the cost. Counting tokens yields a measure that is independent of machines, optimizations, and implementation details. Also, it has been considered as standard practice in the match-algorithm community that time per token is approximately constant [63, 61, 62, 10, 1].[2] So, as a comparative measure of match cost, we will use the number of tokens, in addition to time.

2.2.3 Mapping chunking into a sequence of executable structures

In the sequence of transformations of chunking (Figure 2.10), each intermediate product is non-executable. We can map those non-executable structures into executable structures given appropriate interpreters, as discussed in subsection 2.1.2. Figure 2.12-(b) shows the new sequence of transformations. As the sequence progresses the pseudo-chunks become more like chunks and less like the problem solving. The next chapter illustrates the sequence of transformations and its pseudo-chunks in detail. Also, cost changes through the transformations are analyzed based on the new sequence.

[2] Whenever an overhead (inefficiency) in token processing is found, more efficient algorithms have been developed to eliminate the overhead. For example, a linear list representation of tokens (sequence of WMEs) has been changed to a faster hash table. Also, Rete/UL has introduced elimination of unnecessary processing in the join nodes which maintains constant time per token for learning a large number of rules [9]. The match algorithm employed for this research is a state-of-the-art Rete algorithm without the optimization of Rete/UL. However, the analysis of the number of tokens is still useful when not many rules are learned. Also, we conjecture that the results can be applicable for learning large number of rules if the system employs Rete/UL.

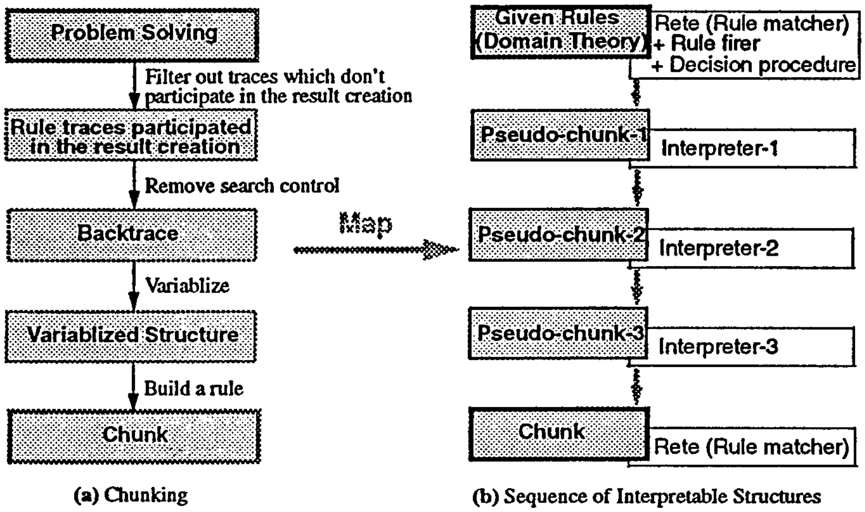

Problem Solving
Filter out traces which don't participate in the result creation
Rule traces participated in the result creation
Remove search control
Backtrace
Variablize
Variablized Structure
Build a rule
Chunk
(a) Chunking
Map
Given Rules (Domain Theory)
Rete (Rule matcher) + Rule firer + Decision procedure
Pseudo-chunk-1
Interpreter-1
Pseudo-chunk-2
Interpreter-2
Pseudo-chunk-3
Interpreter-3
Chunk
Rete (Rule matcher)
(b) Sequence of Interpretable Structures

Figure 2.12: Mapping the sequence of non-executable structures into a sequence of executable pseudo-chunks.

Chapter 3

Transformational Analysis of Chunking

The chunking process can be characterized as a sequence of transformations, and each intermediate product can be mapped into an executable structure (pseudo-chunk) by providing an appropriate interpreter, as described in Chapter 2. This chapter discusses each transformation, and examines the pseudo-chunks, including their effects on cost, with an example. The interpreters provided for pseudo-chunks in the analysis have the same effective measure of cost — number of tokens; the cost of each pseudo-chunk is determined by counting the number of tokens generated during the match to produce the result. By analyzing how the transformations alter the cost, the sources of added expensiveness are revealed.

3.1 Mapping Intermediate Products to Pseudo-chunks

Figure 3.1 shows the mapped sequence of pseudo-chunks from the domain theory to a chunk. This sequence corresponds to the sequence shown in Figure 2.12-(b). Each transformation is the same as the transformations of the chunking process, except for the last step. The last step, building a rule, is divided into two steps in the figure, to examine the match algorithm's restriction on building a new chunk. (Details are given later.)

The pseudo-chunks are generated by mapping the original intermediate products into executable structures by providing an appropriate interpreter. Each pseudo-chunk can be interpreted (by its interpreter) and can produce the same result as the problem solving, given the same initial WMEs. The sequence on the right schematically shows the structural changes through the transformations. This section provides an abstract description of each transformation. (Details will be given in the sections following, along with an example.)

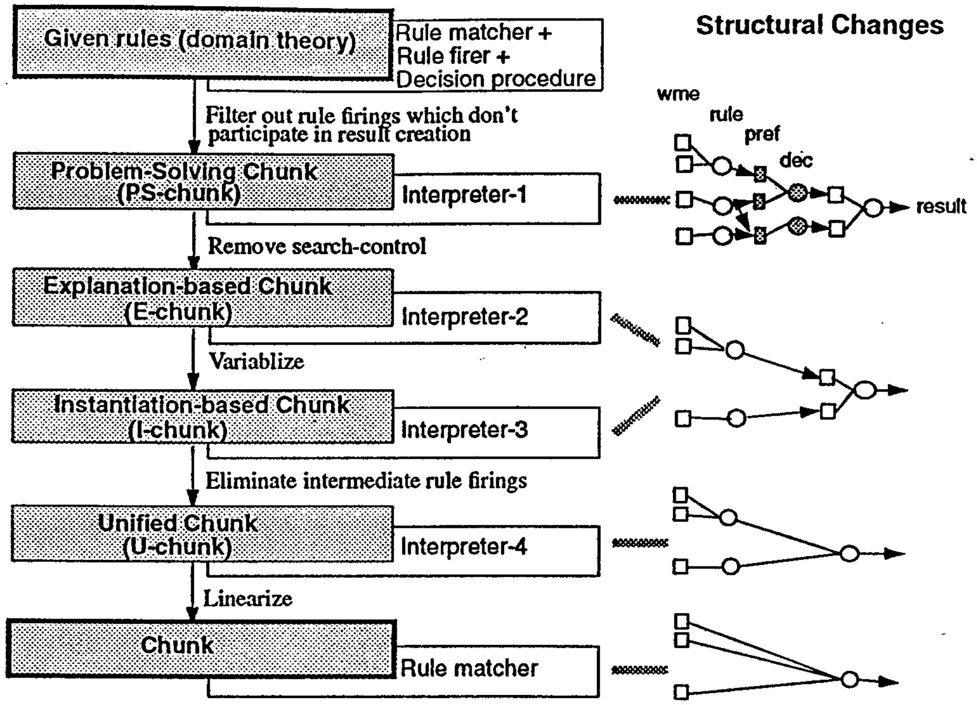

Given rules (domain theory)
Rule matcher +
Rule firer +
Decision procedure
Filter out rule firings which don't participate in result creation
Problem-Solving Chunk (PS-chunk)
Interpreter-1
Remove search-control
Explanation-based Chunk (E-chunk)
Interpreter-2
Variablize
Instantiation-based Chunk (I-chunk)
Interpreter-3
Eliminate intermediate rule firings
Unified Chunk (U-chunk)
Interpreter-4
Linearize
Chunk
Rule matcher
Structural Changes
wme
rule
pref
dec
result

Figure 3.1: A sequence of transformations from the domain theory to a chunk.

The first node in the sequence is the domain theory. The interpreters for the domain theory are the rule matcher, the rule firer, and the decision procedure. The interpretation generates a sequence of rule firings and new WMEs, given the initial WMEs, and produces a problem solving episode.

The first transformation is to eliminate the rule firings that do not participate in the result creation from the problem solving episode. The resulting structure can be mapped to a pseudo-chunk, called a *PS-chunk* (Problem-Solving chunk). Given its interpreter, it reproduces the problem solving episode, excluding unnecessary rule firings.

The second transformation removes search-control traces in the rule traces. Chunking employs only traces from task-definition rules, and omits those from search-control rules, as explained in Section 2.2. After the search-control traces are excluded from the problem solving episode, an *E-chunk* (Explanation-based chunk) is formed from the structure by providing an appropriate interpreter.

The third step is applying the variablization process. The variablization step in chunking is performed by examining the backtrace (explanation), as explained in Section 2.2. The pseudo-chunk generated from the variablized structure is called the *I-chunk* (instantiation-based chunk).

The fourth step is unifying the substructures into a unit. The separate rules in the I-chunk have to be unified into a single structure to produce one rule. This unified structure is called a *U-chunk* (unified chunk).

Finally, the fifth step is to create a new chunk based on the U-chunk. Because the rule compiler (Rete network) requires a linear ordering of the conditions, the graph structure in the U-chunk — which reflects the structure of the rule firings during the problem solving — is linearized into a total ordering, and then conditions are reordered via a heuristic algorithm to improve the match performance.

As noted earlier, the building of a new chunk from the I-chunk is divided into two steps, though the the sequence shown in Figure 2.10 does not distinguish this division. The intermediate step is added to analyze the effect of the rule matcher's restriction on the rule form. Soar's rule matcher, i.e., Rete, assumes a total ordering of the conditions, and the graph structure of an I-chunk is forced to be totally ordered. The intermediate structure (U-chunk) is a unified structure (one rule) that is independent of this restriction. It maintains the graph structure of the I-chunk. By dividing the step of building a new chunk into two sub-steps, we can separate the cost change by unifying (creating one rule from the I-chunk) and the cost change by modifying the rule form to be compiled into Rete.

The following sections describe each pseudo-chunk and its interpreter in detail. These discussions are presented in the context of a simplified Grid task. Figure 3.2 shows a part of the Grid task; that of evaluating if point C is reachable from point A. There are thirty WMEs which record the connections among points and the cars (such as V1 and V2) available to reach the points. In the example, a symbol starting with an upper-case letter is an identifier, and a symbol starting with a lower-case letter is a constant. In this simplified task, the full connections among the points are given only in part. Also, there is only one car available to reach each point. There are four rules to begin within this task. For brevity, rules and WMEs describing Soar's architectural activities are not shown. The details of these activities are given in Chapter 4.

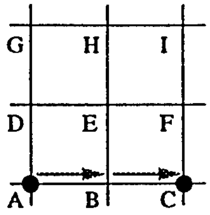
Rule R1 creates a candidate operator whenever there is an adjacent point to the current point. The plus sign (+) in the action indicates that an *acceptable* preference is created for the operator, and it becomes one of the candidate operators. Thus, R1 is one of the task-definition rules that directly propose candidates. Rule R2 can create *best* preferences, which guide the problem solver to pick the operators that go in the right direction. The greater-than sign (>) in the action represents that the operator is the best option among the candidates. Rule R3 applies the selected operator to the state, and changes the current location to the new location indicated by the operator. Finally, rule R4 detects the achievement of the goal by checking if the current location is the same as the given goal point.

According to the EBL specification, the given rules form the domain theory and the given WMEs form a training example. Also, the goal concept is *success* (i.e., the goal point is reachable from the the current position).

3.2 Interpreting the Domain Theory

A problem solving episode, i.e., the input to chunking, can be mapped to the domain theory by providing interpreters. The domain theory is interpreted by the rule matcher, the rule firer, and the decision procedure. The interpretation generates a sequence of rule firings and new WMEs, given the initial WMEs, and produces a problem solving episode. Figure 3.3 shows the sequence of rule firings and WME creations in the problem solving episode. The initial sequence of rule firings of R1 (operator proposal), R2 (search control), and R3 (operator application) is marked as (1) in the figure. The rule firings move the current position from A to B. Also, the subsequent rule firings of R1, R2, and R3 (marked as (2)) moves the current position to C. Finally, R4 detects the achievement of the goal. In the figure, a trivial WME creation is the creation of just one candidate and the creation of a WME from the candidate, as opposed to the WME creations based on other preferences as well as the proposed candidates.

In the graphical representation of the problem solving episode (in Figure 3.3), each circle represents a rule trace, given the WMEs linked to the left of the circle. For example, the leftmost circle in the figure shows that R1 has been fired twice and has created two preferences (P1 and P2). The two preferences propose candidate operators that go to B and to D, respectively. The firings of task-definition rules (rules that directly propose



W1: (A ^next B)
W2: (A ^next D)
W3: (A ^right B)
W4: (B ^next A)
W5: (B ^next C)
W6: (B ^next E)
W7: (B ^right C)
W8: (C ^next F)
W9: (C ^next B)
W10: (D ^next A)
W11: (D ^next E)
W12: (D ^next G)

W27: (G1 ^state S)
W28: (S ^at A)
W29: (G1 ^goal-point GP)
W30: (GP ^at C)

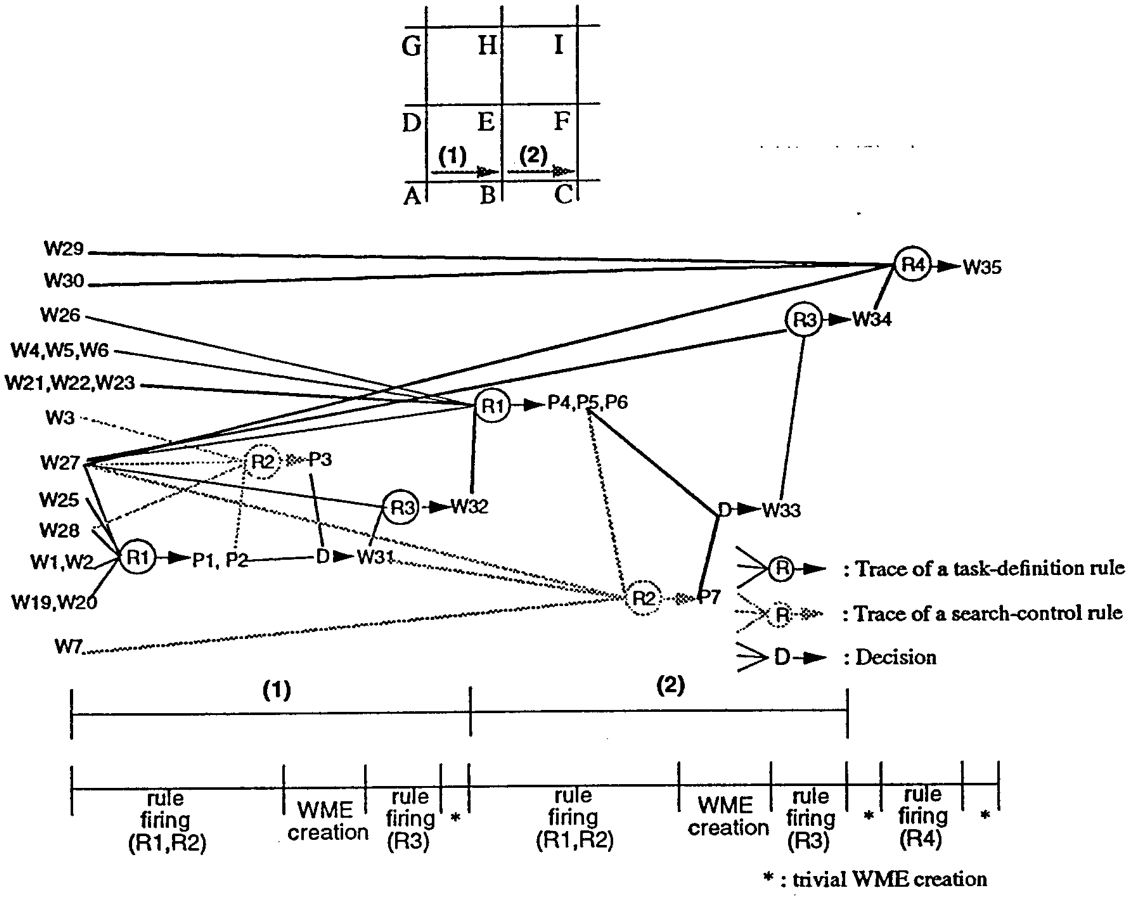
W13: (E ^next B)
W14: (E ^next F)
W15: (E ^next H)
W16: (E ^next D)
W17: (G ^next D)
W18: (G ^next H)
W19: (B ^reachable-by V1)
W20: (D ^reachable-by V1)
W21: (A ^reachable-by V2)
W22: (C ^reachable-by V2)
W23: (E ^reachable-by V2)
W24: (G ^reachable-by V2)
W25: (V1 ^name car)
W26: (V2 ^name car)

(a) Given WMEs

```
(R1                                  ; (operator proposal rule)
(goal <g> ^state <s>)                ; if the location of the current state is
(<s> ^at <loc1>)                     ; <loc1>, and <loc1> is adjacent to <loc2>,
(<loc1> ^next <loc2>)                ; and <loc2> is reachable by some vehicle
(<loc2> ^reachable-by <vehicle>)     ; whose name is <n>, then
(<vehicle> ^name <n>)                ; create a candidate operator to go to <loc2>
-->
(<g> ^operator <loc2> +))

(R2                                  ; (search-control rule)
(goal <g> ^state <s>)                ; if the current location is <loc3>,
(<s> ^at <loc3>)                     ; and <loc4> is on the right, and
(<loc3> ^right <loc4>)               ; there is a candidate operator to go to
(<g> ^operator <loc4> +)             ; <loc4>, then try the operator first than others
-->
(<g> ^operator <loc4> >))

(R3                                  ; (operator application)
(goal <g> ^state <s>)                ; the selected operator goes to <loc5> then
(<g> ^operator <loc5>)               ; change the current location to <loc5>
-->
(<s> ^at <loc5>)

(R4
(goal <g> ^state <s>)                ; (detection of success)
(<g> ^goal-point <gp>)               ; if the current location is <loc6> and
(<gp> ^at <loc6>)                    ; it is the goal point, then the task is accomplished
(<s> ^at <loc6>)
-->
(<s> ^success <loc6>))
```

(b) Given rules

Figure 3.2: A simplified Grid task.



Preferences and WMEs created during problem solving

P1: (G1 ^operator B +)
P2: (G1 ^operator D +)
P3: (G1 ^operator B >)

W31: (G1 ^operator B)
W32: (S ^at B)

P4: (G1 ^operator C +)
P5: (G1 ^operator E +)
P6: (G1 ^operator A +)
P7: (G1 ^operator C >)

W33: (G1 ^operator C)
W34: (S ^at C)
W35: (S ^success C)

Figure 3.3: Problem solving episode excluding unnecessary rule firings.

values of WME) are represented as solid circles, while gray circles indicate search-control rule firings. Each capital letter D represents a synthesis of the preferences by the decision procedure. A connection from one rule to another rule through a decision D, means that preferences created by the former rule are synthesized by a decision to create a WME, and the created WME is matched to a condition of the latter rule. For example, preferences P1, P2, and P3 participate in a decision which creates W31, and W31 is matched to a condition of R3. The trivial decision steps are not shown in the figure for brevity. Also, the acceptable preferences that aren't for the operators are not explicitly represented. Actual interpretation of the domain theory (or problem solving) normally includes other rule firings which are not linked to the result creation; however, these are not shown here.

The details of the match process are shown in Figure 3.4. The matches between the WMEs and the rules are based on the Rete algorithm explained in subsection 2.2.2. A number in front of a rule condition denotes the number of tokens (partial instantiations) generated at that condition in the problem solving episode (as shown in Figure 3.3). The tokens are generated by testing the consistency (*joining*) between the instantiations of the previous conditions and the WMEs matching the current condition (i.e., WMEs in the condition's alpha memory). For example, R1 fires twice in step (1) as follows. The first condition (goal <g> ^state <s>) is instantiated by W27; that is, (W27) is the token of the first condition. Given the token of the first condition, and the instantiations of the second condition (W28 and W30), the consistency test across the two conditions creates one token: (W27,W28). Also, given the token from the first two conditions and the instantiations of the third condition, the consistency test across the first two conditions and the third condition creates two new tokens: (W27,W28,W1) and (W27,W28,W2). Each of them is consistent with one of the instantiations of the fourth condition, and Rete creates two more tokens: (W27,W28,W1,W20) and (W27,W28,W2,W22). Finally, two tokens are created for the last condition: (W27,W28,W1,W20,W25) and (W27,W28,W2,W22,W25). The Rete algorithm creates two instantiations of R1 based on these tokens, and each instantiation creates a new candidate operator by executing the action. The total number of tokens created for these rule firings is 8.

A capital letter S in front of a rule condition indicates sharing of match effort with other rules which have the same patterns of conditions. For example, the instantiations from the first two conditions of rule R1 and the first two conditions of R2 are shared because they have the same patterns of variables tests and constant tests. A part of Rete's efficiency

```
Step (1) -------------------------
(R1
1 (goal <g> ^state <s>)                    (W27)
1 (<s> ^at <loc1>)                         (W27,W28)
2 (<loc1> ^next <loc2>)                    (W27,W28,W1) (W27,W28,W2)
2 (<loc2> ^reachable-by <vehicle>)         (W27,W28,W1,W19) (W27,W28,W2,W20)
2 (<vehicle> ^name <n>)                    (W27,W28,W1,W19,W25) (W27,W28,W2,W20,W25)
-->
2 (<g> ^operator <loc2> CAND))             ==> create P1, P2
(R2
S (goal <g> ^state <s>)                    (W27)
S (<s> ^at <loc3>)                         (W27,W28)
1 (<loc3> ^right <loc4>)                   (W27,W28,W3)
1 (<g> ^operator <loc4> CAND)              (W27,W28,.W3,P1)
-->
1 (<g> ^operator <loc4> BEST))             ==> create P3
Decision process                           P1,P2,P3 ==> create W31
(R3
S (goal <g> ^state <s>)                    (W27)
1 (<g> ^operator <loc5>)                   (W27,W31)
-->
1 (<s> ^at <loc5> ))                       ==> create W32

Step (2) -------------------------
(R1
S (goal <g> ^state <s>)                    (W27)
1 (<s> ^at <loc1>)                         (W27,W32)
3 (<loc1> ^next <loc2>)                    (W27,W32,W4) (W27,W32,W5) (W27,W32,W6)
3 (<loc2> ^reachable-by <vehicle>)         (W27,W32,W4,W21) (W27,W32,W5,W22) (W27,W32,W6,W23)
3 (<vehicle> ^name <n>)                    (W27,W32,W4,W21,W26) (W27,W32,W5,W22,W26)
                                            (W27,W31,W6,W23,W26)
-->
3 (<g> ^operator <loc2> CAND))             ==> create P4, P5,P6
(R2
S (goal <g> ^state <s>)                    (W27)
S (<s> ^at <loc3>)                         (W27,W32)
1 (<loc3> ^right <loc4>)                   (W27,W32,W7)
1 (<g> ^operator <loc4> CAND)              (W27,W32,.W7,P4)
-->
1 (<g> ^operator <loc4> BEST))             ==> create P7
Decision process                           P4,P5,P6,P7 ==> create W33
(R3
S (goal <g> ^state <s>)                    (W27)
1 (<g> ^operator <loc5>)                   (W27,W33)
-->
1 (<s> ^at <loc5>))                        ==> create W34

(R4
S (goal <g> ^state <s>)                    (W27)
1 (<g> ^goal-point <gp>)                   (W27,W29)
1 (<gp> ^at <loc6>)                        (W27,W29,W30)
1 (<s> ^at <loc6>)                         (W27,W29,W30,W33)
-->
1 (<s> ^success <loc6>))                   ==> create W35
```

Figure 3.4: Tokens created during the problem solving.

stems from this type of sharing of the tokens. (The other source of efficiency is the state saving as explained in subsection 2.2.2.) The total cost of the problem solving episode can be computed by summing the number of tokens from the rule firings (irrelevant firings as well as relevant firings), and it is 27 in this case.

3.3 Filtering Out Unnecessary Rule Firings (⇒ PS-chunk)

As a first step toward producing a chunk, unnecessary rule firings that did not participate in the result creation can be filtered out from the problem solving episode. For the given example, this transformation eliminates all other rule firings, if there were any, beyond those shown in Figure 3.5. The resulting structure can be mapped to a PS-chunk by providing an interpreter for it. The interpretation of the resulting PS-chunk looks similar to the original problem solving episode, aside from the missing unnecessary parts. However, its processing differs significantly from the initial problem solving episode in that there are no global memories, such as preference memory and working memory, as well as global buses among the rule firings and WME creations. The interpreter only provides local communications among the rules firings and WME creations based on the problem solving episode. For example, a WME creation (decision) is connected to a rule match only when the original problem solving has a decision where a WME is created by the decision and the created WME is matched to the conditions of the rule. The cycle of rule firings and WME creations in the problem solving episode is linearized into an enclosed sequence of rule firings and WME creations.

In order to build a PS-chunk, for each instance of a rule firing, a copy of the original rule is created. For example, for the first firing of the rule R1, a copy R1-1 is employed. The rules in the PS-chunk are closed off from intermediate WMEs generated outside of this structure. For example, the link between R3-1 and R1-2 through W32 means that no other WMEs except for those created by R3-1 are matched to the condition of R1-2. Also, the WMEs created from R3-1 are not exposed to the matches of other rules. The only parts of a PS-chunk that are exposed to the full set of WMEs, are the conditions matched to the given initial WMEs in the problem solving episode, and the result creation. However, the interpretation of the PS-chunk does not differ in how it uses such optimizations as sharing and state saving. For example, the tokens from the first two conditions of R2-1 are still shared with the tokens from the first two conditions of R1-1. The key difference

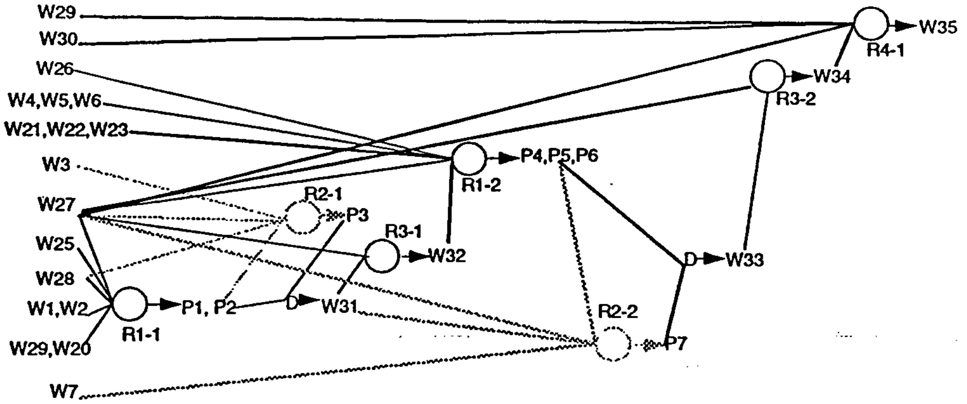


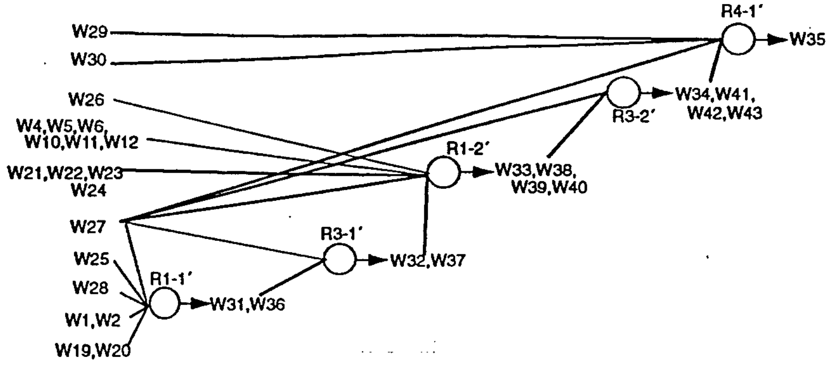
Figure 3.5: A trace of a PS-chunk. A PS-chunk is created by eliminating unnecessary rule firing and encapsulating the problem solving activity into a unit.

between a PS-chunk and a normal chunk is that matching a PS-chunk requires replaying (part of) a problem solving episode (by rule firings and intermediate WME creations), while matching a normal chunk requires just one rule match. Either can create the result in a similar circumstance. We implemented such an interpreter for PS-chunks.

The cost (number of tokens) of a PS-chunk is bounded by the cost of problem solving. If there were unnecessary rule firings in the problem solving, as is usually the case, the cost of a PS-chunk would be strictly less than the cost of the problem solving. Otherwise, the cost would be the same as that of the problem solving. In the given example, there are no unnecessary rule firings, and the cost remains unchanged.

3.4 Removing Search Control (⇒ E-chunk)

Figure 3.5 contains all the rule firings involved in the result creation. However, chunking employs only traces from task-definition rules. The search-control rules are missing in chunking to increase the generality of the learned rules (as explained in Section 2.2). Figure 3.6 shows the E-chunk created from the PS-chunk. The copies of the search-control rule R2 (R2-1 and R2-2) and the nodes representing the decisions are gone, and only the copies of the task-definition rules are maintained in the structure. That is, acceptable preferences are turned directly into WMEs. While interpreting the E-chunk, all candidates proposed by R1-1′ and R1-2′ become WMEs without being filtered by the



WMEs created while matching E-chunk

W31: (G1 ^operator B)	W33: (G1 ^operator C)	W34: (S ^at C)	W35: (S ^success C)
W36: (G1 ^operator D)	W38: (G1 ^operator E)	W41: (S ^at E)	
W32: (S ^at B)	W39: (G1 ^operator A)	W42: (S ^at A)	
W37: (S ^at D)	W40: (G1 ^operator G)	W43: (S ^at G)	

Figure 3.6: A trace of an E-chunk. An E-chunk is created by eliminating search control in a PS-chunk.

search control. This structure can be mapped onto the normal backtrace in chunking (proof tree or explanation in EBL). An E-chunk is identical to an EBL explanation structure. The interpreter for the E-chunk is similar to the interpreter for the PS-chunk except that the former does not have to perform decisions.

The consequence of eliminating search control is that the interpretation of the E-chunk is not constrained by the path actually taken in the problem space, though it will still at least generate the right answer. The interpretation can perform an exponential amount of search even when the original problem-space search is highly directed (by the control rules), as described in Chapter 1. In the above example, without constraining the operator to the best candidate — which goes to the right — the number of tokens in the match of rule R1-2 increases from 10 to 20, as shown in Figure 3.7. Overall, the total number of tokens increases from 27 to 37.

One promising way of avoiding this problem, is to *incorporate search control in chunking* (or just not to drop it) [29]. By incorporating search control in the explanation structure, the match process for the learned rule can focus on the path that was actually followed. The preservation of the search control requires modifications of the subsequent transformations. The details of how to design and implement these modifications are explained in Chapter 5.

```
(1) ----------------------
(R1-1'
1 (goal <g> ^state <s>)                 (W27)
1 (<s> ^at <loc1>)                      (W27,W28)
2 (<loc1> ^next <loc2>)                 (W27,W28,W1) (W27,W28,W2)
2 (<loc2> ^reachable-by <vehicle>)      (W27,W28,W1,W19) (W27,W28,W1,W20)
2 (<vehicle> ^name <n>)                 (W27,W28,W1,W19,W25) (W27,W28,W1,W20,W25)
-->
2 (<g> ^operator <loc2> ))              ==> create W31,W36
(R3-1'
S (goal <g> ^state <s>)                 (W27)
2 (<g> ^operator <loc5>)                (W27,W31) (W27,W36)
-->
2 (<s> ^at <loc5> ))                    ==> create W32,W37

(2) ----------------------
(R1-2'
S (goal <g> ^state <s>)                 (W27)
2 (<s> ^at <loc1>)                      (W27,W32) (W27,W37)
6 (<loc1> ^next <loc2>)                 (W27,W32,W4) (W27,W32,W5) (W27,W32,W6)
                                        (W27,W37,W10) (W27,W37,W11) (W27,W37,W12)
6 (<loc2> ^reachable-by <vehicle>)      (W27,W32,W4,W21) (W27,W32,W5,W22) (W27,W32,W6,W23)
                                        (W27,W37,W10,W21) (W27,W37,W11,W23) (W27,W37,W12,W24)
6 (<vehicle> ^name <n>)                 (W27,W32,W4,W21,W26) (W27,W32,W5,W22,W26) (W27,W32,W6,W23,W26)
-->                                     (W27,W37,W10,W21,W26) (W27,W37,W11,W23,W26) (W27,W37,W12,W24,W26)
4 (<g> ^operator <loc2>))               ==> create W33,W38,W39,W40

(R3-2'
S (goal <g> ^state <s>)                 (W27)
4 (<g> ^operator <loc5>)                (W27,W33) (W27,W38) (W27,W39) (W27,W40)
-->
4 (<s> ^at <loc5> ))                    ==> create W34,W41,W42,W43

(R4-1'
S (goal <g> ^state <s>)                 (W27)
1 (<g> ^goal-point <gp>)                (W27,W29)
1 (<gp> ^at <loc6>)                     (W27,W29,W30)
1 (<s> ^at <loc6>)                      (W27,W29,W30,W34)
-->
1 (<s> ^success <loc6>))                ==> create W35
```

Figure 3.7: Tokens created while matching (interpreting) E-chunk.

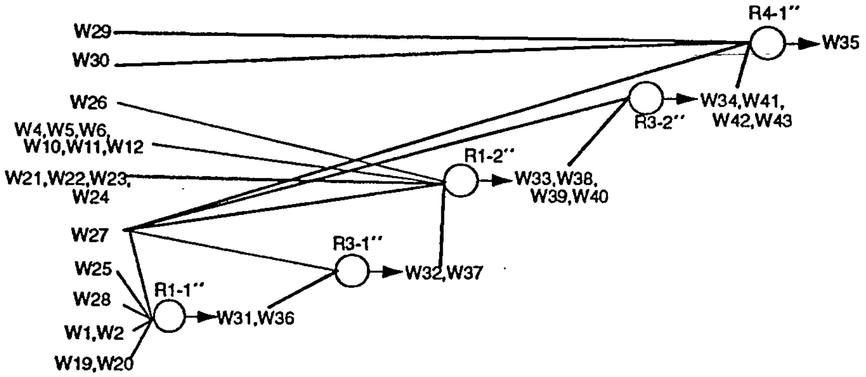
3.5 Variablize (⇒I-chunk)

The variablization step in chunking is performed by examining the backtrace (explanation) that is equivalent to the E-chunk trace. All constants are left alone; they are never replaced by variables. All object identifiers in the instantiations are replaced by variables; in particular, all occurrences of the same identifiers are replaced by the same variable. Since E-chunks consist of rules rather than instantiations, we can model chunking's variablization step as the strengthening of constraints on the match, rather than as the weakening of them. If a variable is instantiated as a constant, it is replaced by that constant. If a variable is instantiated by an identifier, it remains as a variable, but may possibly undergo a name change. Particularly, all variables instantiated by the same identifier are replaced by the same variable. For example, the variables in Figure 3.6 are constrained, as shown in Figure 3.8. The interpreter for the I-chunk is the same as the interpreter for the E-chunk. With the exception of the differences in the variable names, the structures of the I-chunk and the E-chunk are the same.

This transformation can *overspecialize* learned rules when distinct variables in the original rules accidentally happen to match the same identifier. For example, although variable <n1> in R1-1′ and variable <n2> in R1-2′ (Figure 3.6) are instantiated by the same constant **car** and changed to the same constant, they can correctly be generalized as different variables (Figure 3.8). However, from the perspective of cost, this transformation does not increase the number of tokens. The number of tokens generated should remain the same, or be reduced by the introduced constraints. In the given example, the cost remains the same.

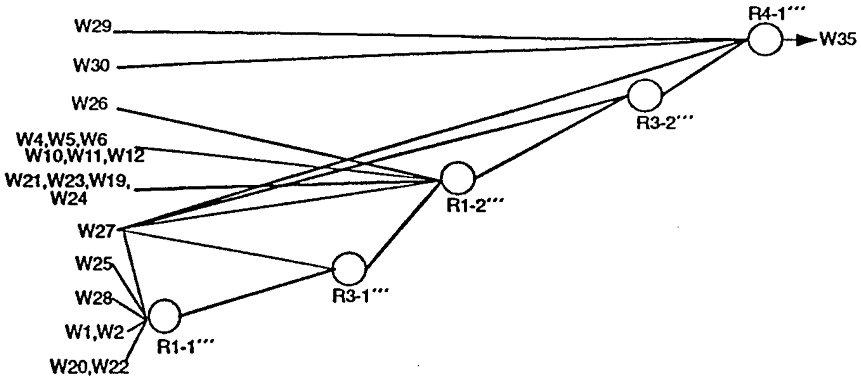
3.6 Eliminating Intermediate Rule Firings (⇒ U-chunk)

This step unifies the separate rules in the variablized structure (I-chunk) into a single rule. Figure 3.9 shows the result of unifying the example I-chunk into the corresponding U-chunk. Although R1-1‴, R3-1‴, R1-2‴, R3-2‴, and R4-1‴ still have their own identifiable conditions in the U-chunk, there are now no intermediate rule firings. The boundaries between the rules are eliminated by removing the intermediate processes of WME creation. In lieu of these processes, the instantiations generated by matching the earlier rules in the firing sequence (i.e., the tokens produced by their final conditions) are



```
(R1-1''
1 (goal <g> ^state <s>)
1 (<s> ^at <loc1>)
2 (<loc1> ^next <loc2>)
2 (<loc2> ^reachable-by <v1>)
2 (<v1> ^name car)
-->
2 (<g> ^operator <loc2> ))

(R3-1''
S (goal <g> ^state <s>)
2 (<g> ^operator <loc2>)
-->
2 (<s> ^at <loc2> ))

(R1-2''
S (goal <g> ^state <s>)
2 (<s> ^at <loc2>)
6 (<loc2> ^next <loc3>)
6 (<loc3> ^reachable-by <v2>)
6 (<v2> ^name car)
-->
4 (<g> ^operator <loc3>))

(R3-2''
S (goal <g> ^state <s>)
4 (<g> ^operator <loc3>)
-->
4 (<s> ^at <loc3> ))

(R4-1''
S (goal <g> ^state <s>)
1 (<g> ^goal-point <gp>)
1 (<gp> ^at <loc3>)
1 (<s> ^at <loc3>)
-->
1 (<s> ^success <loc3>))
```

Figure 3.8: A trace of an I-chunk. An I-chunk is created by constraining variables (by instantiations) in an E-chunk.



```
(R1-1'''
1 (goal <g> ^state <s>)
1 (<s> ^at <loc1>)
2 (<loc1> ^next <loc2>)
2 (<loc2> ^reachable-by <v1>)
2 (<v1> ^name car)

(R3-1'''
S (goal <g> ^state <s>)
2 (R1-1)

(R1-2'''
S (goal <g> ^state <s>)
2 (R3-1''')
6 (<loc2> ^next <loc3>)
6 (<loc3> ^reachable-by <v2>)
6 (<v2> ^name car)

(R3-2'''
S (goal <g> ^state <s>)
6 (R1-2''')

(R4-1'''
S (goal <g> ^state <s>)
1 (<g> ^goal-point <gp>)
1 (<gp> ^at <loc3>)
1 (R3-2''')
-->
1 (<s> ^success <loc3>))
```

Figure 3.9: A trace of a U-chunk. A U-chunk is created by eliminating intermediate rule firings in an I-chunk.

passed directly to the match of the later rules. In effect, this step replaces the intermediate WMEs with the instantiations that created the WMEs. For example, one of R3-1'''’s conditions receives the instantiations of R1-1''' directly as intermediate tokens, rather than receiving WMEs created from the instantiations. Thus, R1-1''', R3-1''', R1-2''', R3-2''', and R4-1''' are no longer (separate) rules. Here, they are called the *subrules*. A condition which matched intermediate WMEs created by a rule in the I-chunk, is replaced by a *nonlinear condition* testing the subrule that is built for the rule. (What makes a condition *nonlinear* is explained in the next paragraph.) When a subrule is tested multiple times by multiple nonlinear conditions, they share the same tokens created for the subrule.

To be able to properly interpret this structure (to measure the cost change through the transformation), an extension is required to the match algorithm. The traditional form of Rete algorithm, as shown in Figure 2.11, requires a *linear* match network, in the sense that a total ordering must be imposed on the conditions to be matched; such as C1, then

C2, and then C3. In (linear) Rete, each *join* node checks the consistency of a token (a partial instantiation) and a WME, with each token itself being a sequence of WMEs, each of which matches one condition. Since the intermediate WMEs are replaced with instantiations, whenever the current condition receives instantiations instead of WMEs, testing the consistency (by a *join* node) between the tokens of previous conditions and the current (nonlinear) condition should *join* two tokens, instead of *joining* a token and a WME. That is, U-chunks require the ability to perform *nonlinear* matches, in which conditions are matched hierarchically via join nodes that compare pairs of tokens, rather than just a single token and a WME. They also require the ability to create hierarchically structured tokens (when pairs of incoming tokens are consistent); that is, a token must now be a sequence of WMEs or tokens (instantiations of a subrule). An extension of Rete, called *nonlinear Rete* [56, 38] has been implemented to interpret this intermediate structure.

An example nonlinear Rete network is given in Figure 3.10. In standard Rete, each right-hand memory is an alpha memory. In nonlinear Rete, the right-hand memory is sometimes a beta memory rather than an alpha memory. For example, the starred right-hand memory is a beta memory containing the instantiations of the subrule R-S1. In R-S2, *joining* between the first condition and the second condition is performed by comparing two tokens instead of a token and a WME. The two optimizations of Rete, sharing and state saving, are still preserved. For example, two subrules can be shared, as long as they have the same pattern of variables and constants. Also, the state saving keeps the previous (partial) matches for use in the future.

Figure 3.11 shows the details of how tokens are created while matching (interpreting) the U-chunk. Instantiations of subrule R1-1‴ are provided as the instantiations of the second condition of R3-1‴. The consistency checking between the instantiations of the first condition and WMEs created by firing R1-1‴ is replaced by a consistency checking between the instantiations of the first condition and the set of instantiations of R1-1‴. This consistency check is based on the common variables between the first condition and the subrule R1-1‴. In this case, there are two common variables, <g> and <s>, and the *join* node checks the equality of the instantiations of these variables. Also, the instantiations of R3-1‴ are provided as the instantiations of the second condition of R1-2‴ in the same way. This process continues until R4-1‴ is instantiated.

```
(R-S1                                  (W26)
1 (goal <g> ^state <s>)                (W26,W27)
1 (<s> ^at <loc1>)                     (W26,W27,W1) (W25,W26,W2)
2 (<loc1> ^next <loc2>)

(R-S2
S (goal <g> ^state <s>)                (W26)
2 (<g> ^operator <loc2>)               (W26, (W26,W27,W1)) (W26,(W26,W27,W1))
    (goal <g> ^state <s>)
    (<s> ^at <loc1>)
    (<loc1> ^next <loc2>)
```

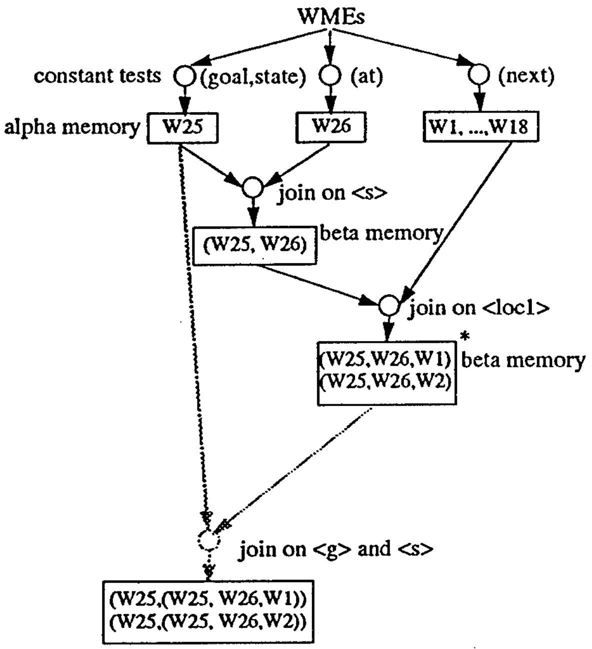


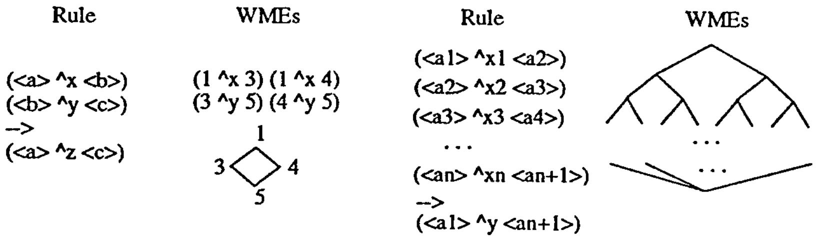
Figure 3.10: An example nonlinear Rete network.

```
(R1-1'''
1 (goal <g> ^state <s>)             (W27)
1 (<s> ^at <locl>)                  (W27,W28)
2 (<locl> ^next <loc2>)             (W27,W28,W1) (W27,W28,W2)
2 (<loc2> ^reachable-by <vl>)       (W27,W28,W1,W19) (W27,W28,W2,W20)
2 (<<vl> ^name car)                 (W27,W28,W1,W19,W25) (W27,W28,W2,W20,W25)

(R3-1'''
S (goal <g> ^state <s>)             (W27)
2 (R1-1''')                         (W27, (W27,W87,W1,W19,W25)) (W27,(W27,W28,W2,W20,W25))

(R1-2'''
S (goal <g> ^state <s>)             (W27)
2 (R3-1''')                         (W27, (W27, (W27,W28,W1,W19,W25)) )
                                     (W27, (W27, (W27,W28,W2,W20,W25)) )
6 (<loc2> ^next <loc3>)             (W27, (W27, (W27,W28,W1,W19,W25)) , W4)
                                      (W27, (W27, (W27,W28,W1,W19,W25)) , W5)
                                      (W27, (W27, (W27,W28,W1,W19,W25)) , W6)
                                      (W27, (W27, (W27,W28,W2,W20,W25)) , W10)
                                      (W27, (W27, (W27,W28,W2,W20,W25)) , W11)
                                      (W27, (W27, (W27,W28,W2,W20,W25)) , W12)
6 (<loc2> ^reachable-by <vl>)       (W27, (W27, (W27,W28,W1,W19,W25)) , W4,W21)
                                      (W27, (W27, (W27,W28,W1,W19,W25)) , W5,W22)
                                      (W27, (W27, (W27,W28,W1,W19,W25)) , W6,W23)
                                      (W27, (W27, (W27,W28,W2,W20,W25)) , W10,W21)
                                      (W27, (W27, (W27,W28,W2,W20,W25)) , W11,W23)
                                      (W27, (W27, (W27,W28,W2,W20,W25)) , W12,W24)
6 (<vl> ^name car)                  (W27, (W27, (W27,W28,W1,W19,W25)) , W4,W21,W26)
                                      (W27, (W27, (W27,W28,W1,W19,W25)) , W5,W22,W26)
                                      (W27, (W27, (W27,W28,W1,W19,W25)) , W6,W23,W26)
                                      (W27, (W27, (W27,W28,W2,W20,W25)) , W10,W21,W26)
                                      (W27, (W27, (W27,W28,W2,W20,W25)) , W11,W23,W26)
                                      (W27, (W27, (W27,W28,W2,W20,W25)) , W12,W24,W26)

(R3-2'''
S (goal <g> ^state <s>)             (W27)
6 (R1-2''')                         (W27,(W27, (W27, (W27,W28,W1,W19,W25)) , W4,W21,W26))
                                      (W27,(W27, (W27, (W27,W28,W1,W19,W25)) , W5,W22,W26))
                                      (W27,(W27, (W27, (W27,W28,W1,W19,W25)) , W6,W23,W26))
                                      (W27,(W27, (W27, (W27,W28,W2,W20,W25)) , W10,W21,W26))
                                      (W27,(W27, (W27, (W27,W28,W2,W20,W25)) , W11,W23,W26))
                                      (W27,(W27, (W27, (W27,W28,W2,W20,W25)) , W12,W24,W26))

(R4-1'''
S (goal <g> ^state <s>)             (W27)
1 (<g> ^goal-point <gp>)            (W27,W29)
1 (<gp> ^at <loc3>)                 (W27,W29,W30)
1 (R3-2''')                         (W27,W29,W30,
                                      (W27, (W27, (W27, (W27,W28,W1,W19),W25) , W5, W22,W26) ) )
```

Figure 3.11: Tokens created while matching U-chunk.

```
(R4-1'''
 (goal <g> ^state <s>)
 (<g> ^goal-point <gp>)
 (<gp> ^at <loc3>)
 (<s> ^at <loc3>)
   (goal <g> ^state <s>)
   (<g> ^operator <loc3>)
      (goal <g> ^state <s>)
      (<s> ^at <loc2>)
         (goal <g> ^state <s>)
         (<g> ^operator <loc2>)
           (goal <g> ^state <s>)
           (<s> ^at <loc1>)
           (<loc1> ^next <loc2>)
           (<loc2> ^reachable-by car)
      (<loc1> ^next <loc3>)
      (<loc2> ^reachable-by car)
-->
 (<s> ^success <loc3>))
```

Figure 3.12: The whole structure of the U-chunk.



(a) An example case of increased tokens

(b) A potential worst case for U-match

Figure 3.13: Number of tokens can increase in a U-chunk.

R4-1''' in Figure 3.12 shows the whole structure of the U-chunk. The level of indentations shows the level in the problem-solving structure. For example, the deepest indented conditions represent R1-1''' which appeared first (leftmost) in Figure 3.9.

Cost problems are introduced in this transformation because the number of instantiations of a rule can be greater than the number of WMEs created from those instantiations. For example, given the rule and WMEs in Figure 3.13-(a), two instantiations — (1 ^x 3) (3 ^y 5) and (1 ^x 4) (4 ^y 5) — are created. Because these two instantiations generate the same bindings for variables <a> and <c>, only one tuple (WME) is generated in the problem solving. Working memory is a set in Soar (and other Ops-like languages), and does not include duplicate elements. Thus, the number of tokens is increased after the WMEs are replaced by the instantiations.

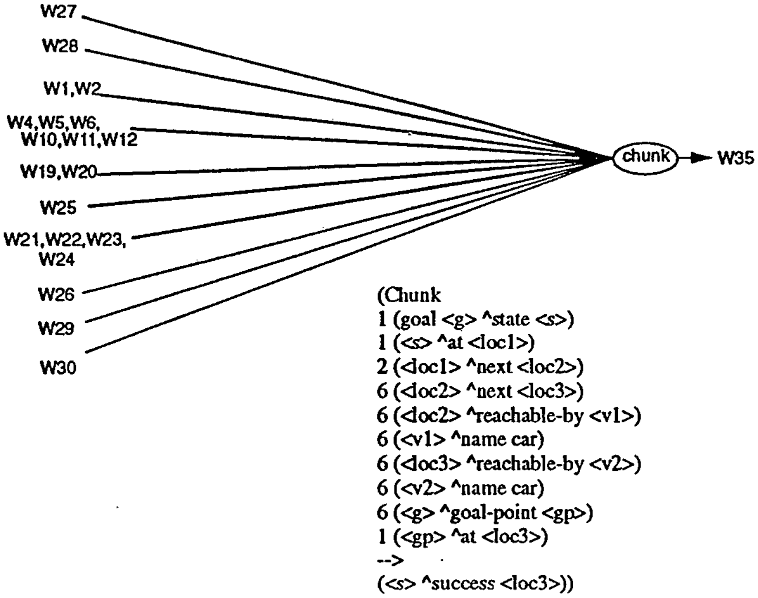
Our grid task also suffers from this problem. In the I-chunk, the six instantiations of R1-2″ create four WMEs since there are only four points that can be reached by moving two steps from A. The four WMEs are then matched to the second condition of R3-2″. However, in the U-chunk, the six instantiations are directly used, and create two more tokens. This increases the total number of tokens from 37 to 39. A worst case can arise when the working memory is structured as in Figure 3.13-(b). While the number of instantiations is exponential in the number of conditions, the number of WMEs is only one.

Our proposed solution to this problem is to *preprocess instantiations before they are used* so that the number of tokens passed from a substructure of a U-chunk is no greater than the number of WMEs passed in the corresponding I-chunk. This could potentially be done either by grouping instantiations that generate the same WME or by selecting one of them as a representative. The details of this solution and the impact of this modification to the subsequent transformations are given in Chapter 5.

3.7 Linearizing (⇒ Chunk)

As described in subsection 2.2.2, after a chunk is created, the operational conditions are compiled into a Rete network for future matches of the learned rule. In the process, the hierarchy in the U-chunk (which reflects the structure of the rule firings during problem solving) is linearized into a total ordering. Conditions are then reordered via a heuristic algorithm to improve the match performance. For example, the nonlinear structure in Figure 3.9 can be linearized to the structure in Figure 3.14.

The critical consequence of this step (linearization and condition ordering) is that the match structure of the learned rule is no longer constrained by the problem-solving structure. That is, how instantiations of different conditions are combined, can be different from how they were combined during the problem solving. This structural change introduces three different sources of expensiveness. The first source arises directly from the linearization of the graph structure. By combining sub-graphs (of the subrules) together, some of the previously independent conditions become *joined* with other parts of the structure before they finish their sub-hierarchy match. Figure 3.15 shows an example. Figure 3.15-(b) shows the rule firing structure during the problem solving, given the WMEs and rules in Figure 3.15-(a). The structures of the linearized rules are shown



```
(Chunk
1 (goal <g> ^state <s>)
1 (<s> ^at <loc1>)
2 (<loc1> ^next <loc2>)
6 (<loc2> ^next <loc3>)
6 (<loc2> ^reachable-by <v1>)
6 (<v1> ^name car)
6 (<loc3> ^reachable-by <v2>)
6 (<v2> ^name car)
6 (<g> ^goal-point <gp>)
1 (<gp> ^at <loc3>)
-->
(<s> ^success <loc3>))
```

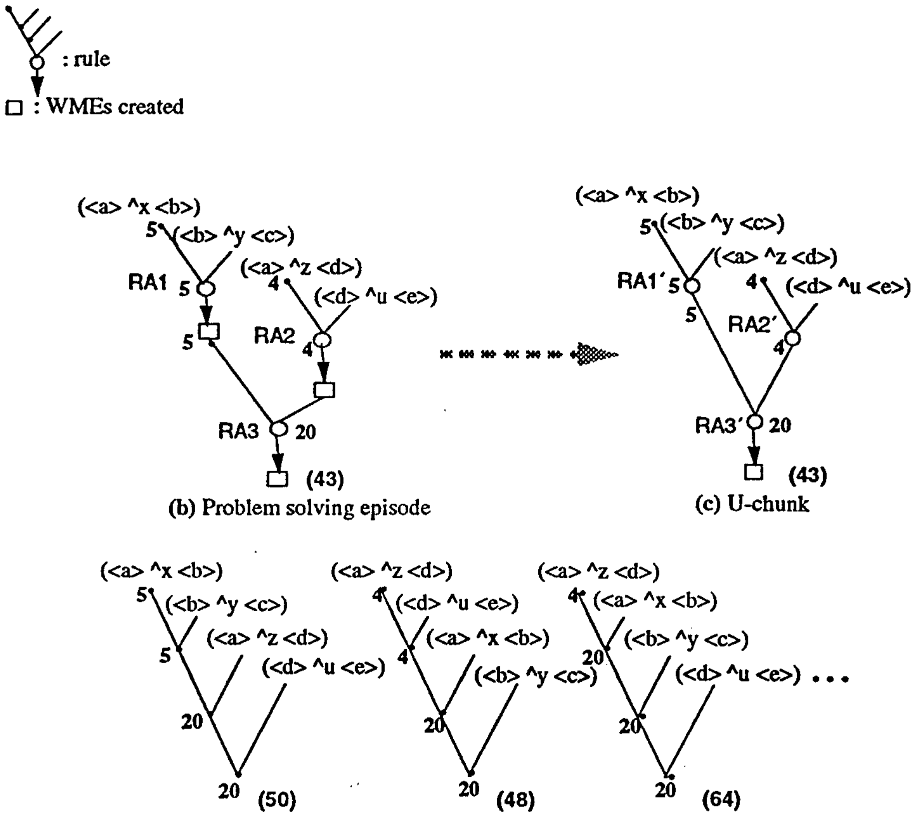
Figure 3.14: Chunk: results from linearizing the U-chunk.

explicitly in Figure 3.15-(d). The number in front of each node indicates the number of tokens at that condition. The total number of tokens in the match for the rule is the sum of these numbers (43 in this case). The U-chunk created from the problem solving episode is shown in Figure 3.15-(c). In the problem-solving episode and the U-chunk, the conditions in a subrule (e.g., the conditions in RA1) are matched independently from the other parts of the structure (e.g., the conditions of RA2) before its created WMEs are *joined* with the WMEs created by RA2. By combining these sub-graphs together — through linearization — some of these previously independent conditions are *joined* with other parts of the structure before they finish their sub-graph match. In Figure 3.15-(d), it is no longer possible to maintain independence between the conditions of RA1 and RA2. For example, in the first case, tokens for the conditions from RA2 — (<a> ^z <d>) and (<d> ^u <e>) — are dependent on tokens for the conditions of RA1.

This *loss of independence* can increase the number of tokens. For the three orderings shown in Figure 3.15-(c), the number of tokens for the linearized structures are 50, 48, and 64, which are all greater than 43. No matter what condition ordering is used, the number of tokens still increases, given the WMEs in Figure 3.15-(a).

```
(a1 ^x b1) (a1 ^x b2) (a1 ^x b3) (a1 ^x b4) (a1 ^x b5)
(b1 ^y c1) (b2 ^y c2) (b3 ^y c3) (b4 ^y c4) (b5 ^y c5) (b6 ^y c6)
(a1 ^z d1) (a1 ^z d2) (a1 ^z d3) (a1 ^z d4)
(d1 ^u e1) (d2 ^u e2) (d3 ^u e3) (d4 ^u e4) (d5 ^u e5)

RA1)             RA2)             RA3)
(<a> ^x <b>)     (<a> ^z <d>)     (<a> ^k <c>)
(<b> ^y <c>)     (<d> ^u <e>)     (<a> ^l <e>)
-->              -->              -->
(<a> ^k <c>)     (<a> ^l <e>)     (<a> ^is success)
```

(a) Working memory elements and rules



(b) Problem solving episode

(c) U-chunk

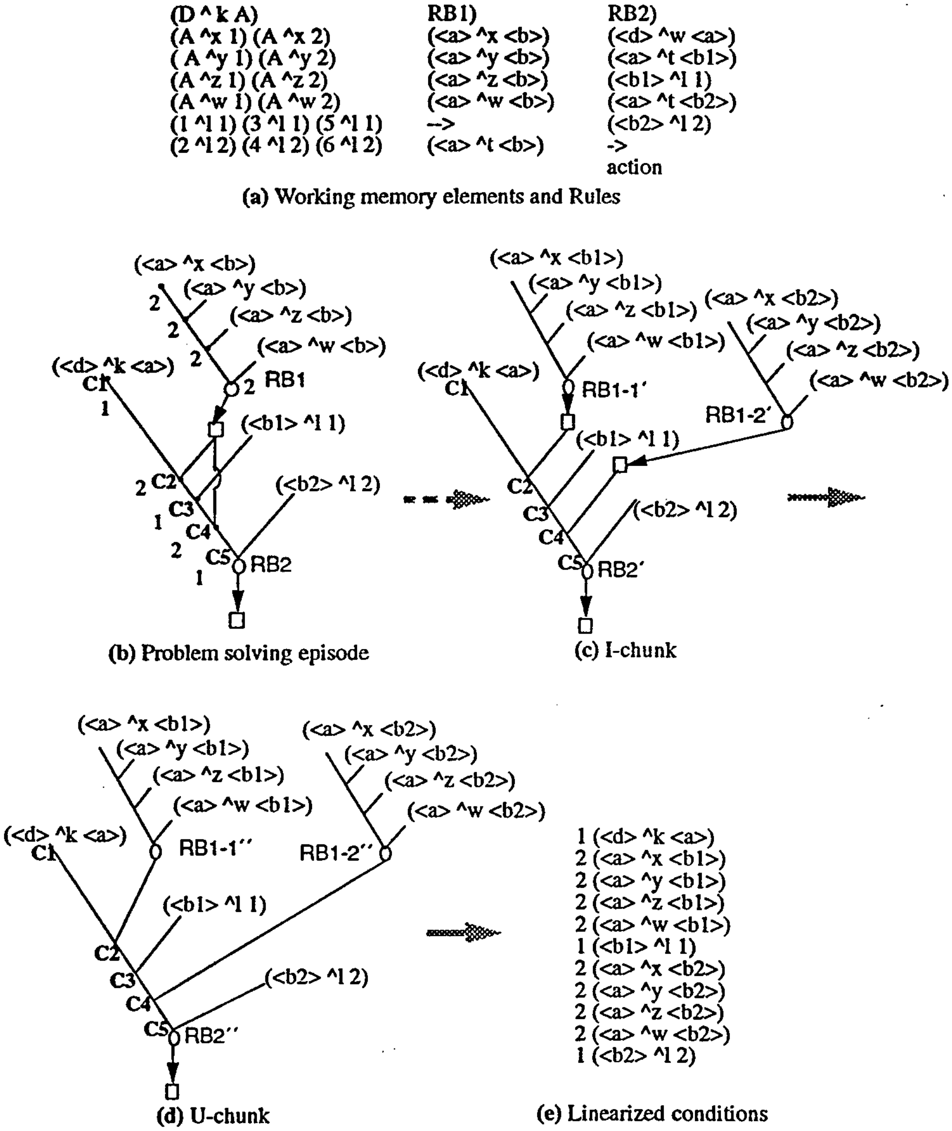
(d) Possible linearized structures of (c)

Figure 3.15: Loss of independence by linearization.

The second source of cost increase is *loss of sharing*. As long as Rete cannot capture the sharing from the nonlinear structure, the number of tokens can increase. Figure 3.16 shows an example. Given the rules in Figure 3.16-(a), the problem solving shares the instantiations of RB1 for both conditions C2 and C4 of rule RB2. That is, they match the WMEs created from the instantiations of RB1. (The total number of tokens is 15 in the problem solving.) Although the instantiations are shared, C2 and C4 are matched by different WMEs because <b1> and <b2> cannot be bound to the same value (given the initial set of WMEs in Figure 3.16-(a)). Thus, two instantiations of RB1 participate in the backtrace (explanation); one of them creates the WME matched by C2, and the other creates the WME matched by C4. Figure 3.16-(c) shows the I-chunk generated from the backtrace. RB1 is separated into RB1-1′ and RB1-2′, by replacing the two instantiations with two rules. Although they are separated, the two subrules have the same network structure and the same pattern of consistency tests across the conditions, and they can be compiled into the same structure. The total cost remains as 15. The U-chunk created from the I-chunk is shown in Figure 3.16-(d). Instantiations of RB1-1″ and RB1-2″ can still be shared in nonlinear Rete as subrules with the same patterns are shared in the network. The total cost of the U-chunk is also 15. The chunk (with an optimal ordering) generated from the U-chunk is shown in Figure 3.16-(d). Linearization loses the structural information, so the sharing of sub-parts becomes impossible. The total number of tokens is increased from 15 to 19.

The third source of cost increase comes from *non-optimal ordering* of the conditions. The computational complexity of finding an optimal ordering for a set of conditions is a factor of the factorial in the number of conditions (considering all possible orderings), so Rete employs a heuristic ordering algorithm. Because the heuristic condition-ordering algorithm cannot guarantee optimal orderings, whenever this algorithm creates a *non-optimal ordering*, additional cost may be incurred. For example, our Grid task can create the non-optimally-ordered chunk shown in Figure 3.14. The cost is increased from 39 to 41 with this chunk. However, with an optimal ordering, as shown in Figure 3.17, the cost can be reduced to 11.

Our proposed solution to this set of problems is to *eliminate the linearization step*. By keeping the graph structure — that is, by replacing chunks with U-chunks — all three causes of cost increase can be avoided. The key condition that this requires is an efficient generalization of Rete for nonlinear match, as shown in Figure 3.10.



```
(D ^ k A)                      RB1)              RB2)
(A ^x 1) (A ^x 2)              (<a> ^x <b>)      (<d> ^w <a>)
( A ^y 1) (A ^y 2)             (<a> ^y <b>)      (<a> ^t <b1>)
(A ^z 1) (A ^z 2)              (<a> ^z <b>)      (<b1> ^l 1)
(A ^w 1) (A ^w 2)              (<a> ^w <b>)      (<a> ^t <b2>)
(1 ^l 1) (3 ^l 1) (5 ^l 1)     -->               (<b2> ^l 2)
(2 ^l 2) (4 ^l 2) (6 ^l 2)     (<a> ^t <b>)      ->
                                                 action
```

(a) Working memory elements and Rules

(b) Problem solving episode

(c) I-chunk

(d) U-chunk

1 (<d> ^k <a>)
2 (<a> ^x <b1>)
2 (<a> ^y <b1>)
2 (<a> ^z <b1>)
2 (<a> ^w <b1>)
1 (<b1> ^l 1)
2 (<a> ^x <b2>)
2 (<a> ^y <b2>)
2 (<a> ^z <b2>)
2 (<a> ^w <b2>)
1 (<b2> ^l 2)

(e) Linearized conditions

Figure 3.16: Loss of sharing by linearization.

```
(Chunk
1 (goal <g> ^state <s>)
1 (<g> ^goal-point <gp>)
1 (<gp> ^at <loc3>)
1 (<s> ^at <loc1>)
2 (<loc1> ^next <loc2>)
1 (<loc2> ^next <loc3>)
1 (<loc2> ^reachable-by <v1>)
1 (<v1> ^name car)
1 (<loc3> ^reachable-by <v2>)
1 (<v2> ^name car)
-->
(<s> ^success <loc3>))
```

Figure 3.17: The match cost of an optimally ordered chunk.

3.8 Summary

The above sections have described an analysis of the chunking process as a sequence of transformations in which each intermediate product is mapped into a pseudo-chunk by providing an appropriate interpreter. By computing and comparing the cost of the pseudo-chunks, we have identified a set of sources that can make the output chunk expensive. In addition to identifying which transformations lead to cost increases, and how they lead to such increases, the analysis has also pointed the way toward modifications of the transformational sequence that could potentially eliminate these cost increases. The set of sources and the proposed modifications are :

1. *Removing search control ⇒ incorporate search control in chunking.* By incorporating search control in the explanation structure, the match process for the learned rule can focus on the path that was actually followed.

2. *Disrupting the optimizations based on equivalent knowledge ⇒ preprocess knowledge before it is used.* By preprocessing the tokens, the number of tokens passed from a substructure of a U-chunk can be no greater than the number of WMEs passed in the corresponding I-chunk. This could potentially be done either be grouping instantiations that generate the same WME or by selecting one of them as a representative. These optimizations are called *token compression.*

3. *Linearizing (Losing efficiencies stemming from problem-solving structures) ⇒ keep the problem-solving structure.* By keeping the problem-solving structure — that

is, by replacing chunks with U-chunks — all three causes of cost increase can be avoided.

To be able to more easily generalize the above analysis to other EBL systems, the next chapter performs a transformational analysis of an EBL implementation in Soar (Soar/EBL).

Chapter 4

Transformational Mapping of EBL onto Soar

In past work, chunking in Soar has been analyzed as a variant of EBL. The four components (the goal concept, the training example, the domain theory, and the operationality criterion) and sub-processes of EBL have been mapped to the components of Soar and to the sub-processes of chunking, respectively [53]. Also, the cost and the generality of the learned rules have been compared [67]. We extend this earlier work by first implementing EBL within Soar (Version 6) — to yield Soar/EBL — and then analyzing this implementation as a sequence of transformations from a problem solving episode to a learned rule. Each intermediate product (and the input and the output) can be mapped into a pseudo-chunk, which can be evaluated with respect to its cost, and thus directly compared in terms of cost against both the original problem solving and the ultimate *EBL-rule*. The results of this analysis are compared with the transformational analysis of chunking presented in Chapter 3.

These discussions are presented in the context of the cup domain [45] — a typical illustrative EBL task. The cup domain representation, shown in Figure 4.1-(a), is an extension of the domain rules shown in Section 2.1. Some conditions in the rules given here test results from Soar's architectural activities. In Soar, some problem solving activities do not involve rule firings. For example, the architectural activities — including the acts of signaling that an impasse has occurred and creating a subgoal — are performed by the architecture itself, not by firings of the rules. These activities can create *architectural WMEs*, such as W1 (supergoal-subgoal relationship) and W2 (the impasse type). These architectural WMEs are tested by the rules. Thus, the actual domain theory is close to the rules shown in Figure 4.1. The transformational analysis of chunking in Chapter 3 has concentrated on the transformations of the rule firings and the decisions, and excluded this architecture related aspect. This chapter examines the transformations of the architectural activities through the learning process, as well as the transformations of the rule firings and decisions, and analyzes how they affect the output rule.

In the domain theory (Figure 4.1-(a)), R1 can create a new problem space named "cup" and a new state, given the lack of information in the supergoal situation about which object is a cup. In R2, R3, and R4, the training example is accessed through the attribute **super-state**, which links the cup problem-space state and the supergoal state. The training example (i.e., the supergoal situation) is illustrated in Figure 4.1-(b), as the WMEs that existed before the subgoal process.

Figure 4.2-(a) shows the two sequences of transformations that represent chunking and Soar/EBL. The chunking part is the same as the transformations introduced (in Figure 3.1) in Chapter 3. Each intermediate product is mapped into a pseudo-chunk in chunking and Soar/EBL. By comparing the two sequences, we can clarify the relationship between the two systems. Also, by analyzing how the transformations alter cost and generality, a set of sources of added expensiveness and changes in generality can be contrasted.

The following sections analyze the transformations underlying Soar/EBL, along with their resulting pseudo-chunks and their effects on cost and generality. This analysis is then compared with the results from the corresponding transformations and intermediate results in chunking. Examples are taken from the cup domain (Figure 4.1).

4.1 Filtering Out Unnecessary Rule Firings (⇒PS-chunk)

As in chunking, a problem solving episode can be mapped to the domain theory by providing its interpreters (the rule matcher, the rule firer, and the decision procedure). The interpretation generates a problem solving episode. It corresponds to the EBL step of "using the domain theory to prove that the training example is an instance of the goal concept".

The first transformation applies to this episode, and filters out any rule firings which did not participate in creating the result. The resulting pseudo-chunk is the same as the PS-chunk described in Chapter 3. In the cup example, this transformation eliminates all other rule firings, if there were any, beyond those shown in Figure 4.3. The interpreter linearizes the cycles of the rule firings and decisions in the problem solving into an enclosed sequence of rule firings and decisions. Its implementation incorporates the same

```
(R1
(goal <g> ^impasse no-change)
(<g> ^super-goal <sg>)
(<sg> ^state <ss>)
-->
(<g> ^problem-space <p> +)
(<g> ^state <s> +)
(<p> ^name cup +)
(<s> ^super-state <ss> +))
```

```
(R2
(goal <g> ^problem-space <p>)
(<p> ^name cup)
(<g> ^state <s>)
(<s> ^super-state <ss>)
(<ss> ^object <o>)
(<ss> ^part-rel <pr>)
(<pr> ^part-of <o>)
(<o> ^is light)
(<pr> ^part <hd>)
(<hd> ^isa handle)
-->
(<s> ^liftable <o> +))
```

```
(R3
(goal <g> ^problem-space <p>)
(<p> ^name cup)
(<g> ^state <s>)
(<s> ^super-state <ss>)
(<ss> ^object <o>)
(<ss> ^part-rel <pr>)
(<pr> ^part-of <o>)
(<pr> ^part <bt>)
(<bt> ^isa bottom)
(<bt> ^is flat)
-->
(<s> ^stable <o> +))
```

```
(R4
(goal <g> ^problem-space <p>)
(<p> ^name cup)
(<g> ^state <s>)
(<s> ^super-state <ss>)
(<ss> ^object <o>)
(<ss> ^part-rel <pr>)
(<pr> ^part-of <o>)
(<pr> ^part <conc>)
(<conc> ^isa concavity)
(<conc> ^is upward-pointing)
-->
(<s> ^open-vessel <o> +))
```

```
(R5
(goal <g> ^problem-space <p>)
(<g> ^super-goal <sg>)
(<p> ^name cup)
(<g> ^state <s>)
(<s> ^open-vessel <o>)
(<s> ^liftable <o>)
(<s> ^stable <o>)
-->
(<o> ^isa cup +))
```

(a) Domain theory

```
W1 : (G2 ^super-goal G1)
W2 : (G2 ^impasse no-change)
W3 : (G1 ^state S1)
W4 : (S1 ^object O1)
W5 : (S1 ^own-rel Relation-1)
W6 : (S1 ^part-rel Relation -2)
W7 : (O1 ^is light)
W8 : (Relation-1 ^owner Edgar)
W9 : (Relation-1 ^owned O1)
W10:(Relation-2 ^part-of O1)
W11:(Relation-2 ^part Concavity-1)
W12:(Relation-2 ^part Handle-1)
W13:(Relation-2 ^part Flat-bottom-1)
W14:(Handle-1 ^isa handle)
W15:(Flat-bottom-1 ^isa bottom)
W16:(Flat-bottom-1 ^is flat)
W17:(Concavity-1 ^isa concavity)
W18:(Concavity-1 ^is upward-pointing)
```

(b) Training Example (Given WMEs before subgoal processing)

Figure 4.1: Cup domain in Soar.

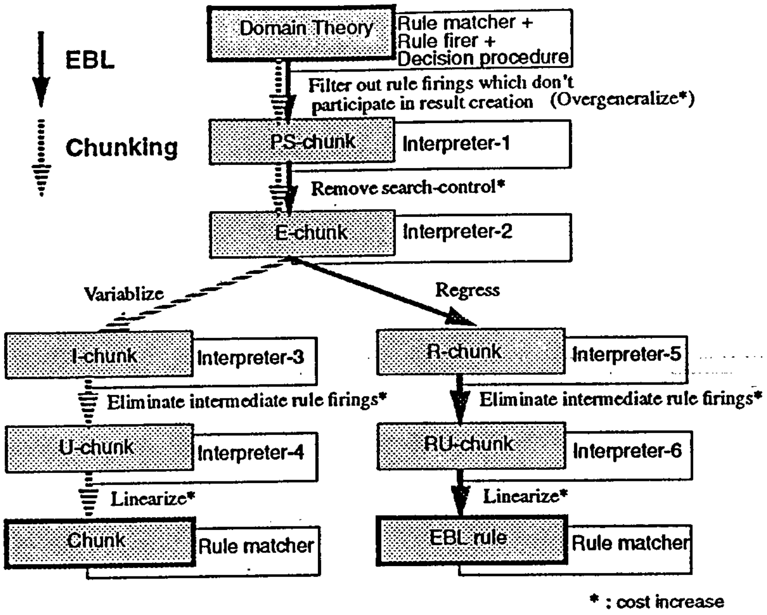
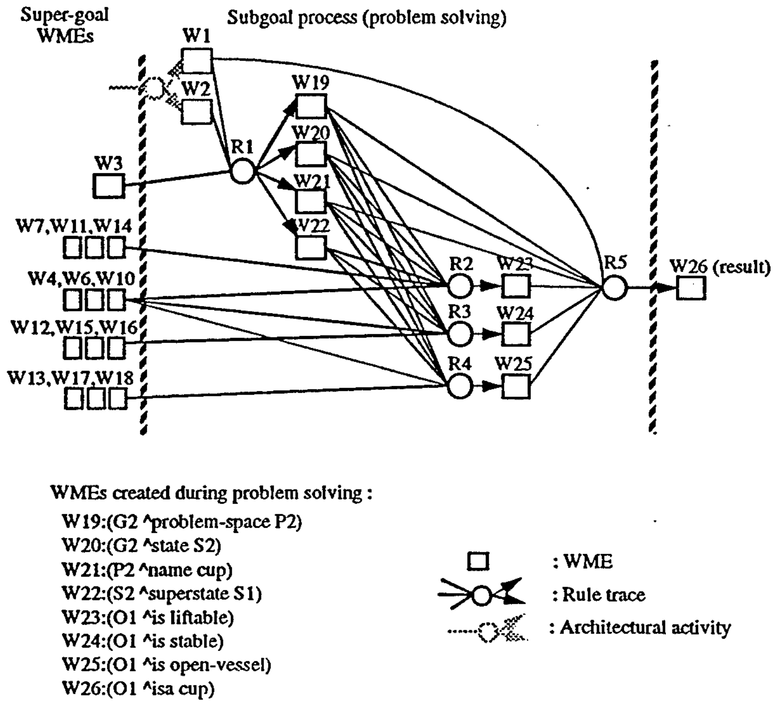


Figure 4.2: The transformational sequences underlying chunking and Soar/EBL.

optimizations that are used in the original problem solving; for example, the tokens from the first four conditions of R3 and R4 are still shared with the tokens from the first four conditions of R2 in the match network for the PS-chunk.

The architectural activities are represented as gray circles with lines attached to them. Because these activities are not represented as rule traces in Soar, they can leave holes in the backtrace. Also, non-operational negated conditions test the absence of objects in the subgoal, and this test has no trace to the supergoal elements. So, Soar implicitly provides two architectural axioms that model these activities, much as in [41]. First, if a WME is obviously based on a supergoal object, a dummy instantiation that links them is created and added to the backtrace. Second, if it is intractable to compute the linkage to supergoal objects, the backtrace simply ignores the WME — just as if it had been created by a rule with no conditions. For example, to operationalize the non-operational negated conditions, we have to analyze why there is no WME matching the conditions, and this requires exhaustive examination of all the rules in the system to find out which rules might have created WMEs that can be matched to the negated conditions. Because of this intractability, non-operational negated conditions are ignored in learning. This may yield overgeneralization, but in return, it helps maintain tractability. The use of these



WMEs created during problem solving :

W19:(G2 ^problem-space P2)
W20:(G2 ^state S2)
W21:(P2 ^name cup)
W22:(S2 ^superstate S1)
W23:(O1 ^is liftable)
W24:(O1 ^is stable)
W25:(O1 ^is open-vessel)
W26:(O1 ^isa cup)

Figure 4.3: Problem solving episode excluding unnecessary rule firings. This structure embodies a PS-chunk in both chunking and Soar/EBL.

```
Given:
        W1: (G2 ^super-goal G1) ; G2 is a subgoal of G1
        W2: (G2 ^impasse no-change) ; G2 is created because of a no-change impasse

Create a dummy rule:
          (dummy
           <g1> ; test the goal
           -->
          (<g2> ^object <g1>); <g2> is a subgoal of <g1>
          (<g2> ^impasse no-change) ; create a no-change impasse
```

Figure 4.4: Creation of a dummy rule to interpret the architectural activities.

architectural axioms is driven by the nature of Soar's architectural activities and intractable negated conditions, and is independent of whether learning occurs via chunking or EBL. Thus, Soar/EBL and chunking share this source of overgenerality.

For the architectural activities of the PS-chunk shown in Figure 4.3, the second axiom (ignoring the WMEs) is applied. To interpret these activities as a part of the PS-chunk match, a dummy rule is introduced.[1] Figure 4.4 shows the dummy rule, which can produce the same WMEs as those created by the architectural activities in the problem solving episode. The dummy rule tests only a supergoal, and creates two new WMEs. This dummy rule introduces the same source of overgenerality to the learned rule as Soar does. The creation of W1 and W2 in the problem solving episode is based on the architectural detection of certain conditions — a no-change impasse. The impasse arose because there was not enough information to make any progress in the supergoal. Here, we produce the same effect without detecting such conditions.

The overgenerality caused by using the architectural axioms can also lead to cost changes. For example, the above dummy rule can be matched to any goal, as well as the goal that detected the impasse. Thus, the match cost can increase to match other goals, depending on the number of goals in the system. This aspect should be combined with the analysis of non-architectural activities, but it is left as a future work.

Except for the changes caused by the architectural activities, there is no other source of cost increase. Because PS-chunks are created by filtering out unnecessary rules in problem solving, and their implementation preserves the match optimizations, the transformation itself does not increase the cost. If there were unnecessary rule firings in the problem

[1] This dummy rule is only in the analysis, not in the implementation.

solving (as is usually the case), the transformation decreases the cost. Otherwise, the cost is identical modulo overgenerality.

4.2 Removing Search Control (⇒ E-chunk)

This step removes search control (if there is any) from a PS-chunk. PS-chunks incorporate all rules which are linked to the result creation. That is, they include both task-definition rules and search-control rules. However, in archetypical EBL systems, implemented for Prolog-like languages, the problem solving does not employ search-control rules. Even in Prodigy/EBL, where the problem solving involves search control, the explanation ignores some of its search-control rules [43]. When a problem solving episode includes search-control rule firings, but the EBL system ignores them in the learning process, the cost of the learned rule can increase as explained in Chapter 1 and Section 3.1. The learned rule (and the pseudo-chunks created between the transformation and the learned rule) are not constrained by the search control, and can therefore perform an exhaustive search, even when the original search was highly directed.

Given that the PS-chunk is shared by both systems and both are transformed in the same way, the resulting pseudo-chunk is the same as the E-chunk. Because no search control is used in the simple cup domain, the structure of the E-chunk is the same as the PS-chunk shown in Figure 4.3. However, if the problem solving employs search-control rules, as shown in Section 3.4, this step can increase the cost. The E-chunk acts as an EBL explanation structure.

4.3 Regressing (⇒R-chunk)

The next step in EBL is regression. Replacing the variable names with unique names (building the explanation structure) and then unifying each connection between an action and a condition, can create a generalized explanation from the explanation. We build the explanation structure by examining the E-chunk trace that is equivalent to the explanation, and applying the regression process of [46] to the explanation structure.

Soar/EBL needs to introduce some additional constraints on variable names in order to produce legal Soar rules. For example, since one goal cannot have more than a single supergoal, allowing multiple variable names for the supergoal leads to superfluous — i.e., unusable — generality, while also possibly leading to legality problems. The *R-chunk* (regressed chunk) resulting from the combination of regression along with these additional variable constraints is shown in Figure 4.5-(a). In this example, the structure remains the same as in the E-chunk.

As shown by the divergence in Figure 4.2, chunking performs a different transformation. The variablization step in chunking is performed by examining the backtrace (explanation) instead of the explanation structure, as described in Section 3.3. For example, the variables in the E-chunk can be constrained to the I-chunk as shown in Figure 4.5-(b). One advantage of this form of instantiation-based constraining over regression (in Soar), is that it naturally introduces the required architectural constraints. For example, the value field of the second condition of R1 and the second condition of R5 are bound to the same identifier G1, and are replaced by the same variable. As long as the instantiations reflect the architectural constraints, the I-chunk automatically preserves them.

However, an I-chunk can be overspecialized, as explained in Chapter 3. For example, although variable <pr> in R2 and variable <pr> in R3 (Figure 4.1) are instantiated by the same identifier Relation-2, and changed to the same variable <p2>, they can be correctly generalized as different variables, as in Figure 4.5-(a). Regression also maintains relational tests among the variables bound to the constant, where chunking explicitly replaces them by constants.

The interpreter for the R-chunk is the same as the interpreter for the E-chunk. Except for the differences in the variable names, the structures of the R-chunk and the E-chunk are identical. With respect to the cost, regression does not increase the number of tokens. The number of tokens should remain the same, or be reduced by the extra constraints.

4.4 Eliminating Intermediate Rule Firings (⇒ RU-chunk)

This step unifies the separate rules of the R-chunk into a single rule, called a *RU-chunk* (regressed-and-unified chunk), as chunking unifies an I-chunk into a U-chunk. Figure 4.6 shows the result of unifying the R-chunk in Figure 4.5-(a) into the corresponding RU-chunk. Intermediate WMEs are replaced with the instantiations which created the WMEs. Although R1-2, R2-2, ..., R5-2 still have their own identifiable conditions in the RU-chunk, there are now no intermediate rule firings. For example, one of R5's conditions

```
(R1-1
(goal <g1> ^impasse no-change)
(<g1> ^super-goal <g2>)
(<g2> ^state <s2>)
->
(<g1> ^problem-space <p1> +)
(<g1> ^state <s1> +)
(<p1> ^name cup +)
(<s1> ^super-state <s2> +)
```

```
(R2-1
(goal <g1> ^problem-space <p1>)
(<p1> ^name cup)
(<g1> ^state <s1>)
(<s1> ^super-state <s2>)
(<s2> ^object <o1>)
(<s2> ^part-rel <p6>)
(<p6> ^part-of <o1>)
(<o1> ^is light)
(<p6> ^part <h1>)
(<h1> ^isa handle)
->
(<s1> ^liftable <o1> +))
```

```
(R3-1
(goal <g1> ^problem-space <p1>)
(<p1> ^name cup)
(<g1> ^state <s1>)
(<s1> ^super-state <s2>)
(<s2> ^object <o1>)
(<s2> ^part-rel <p8>)
(<p8> ^part-of <o1>)
(<p8> ^part <b1>)
(<b1> ^isa bottom)
(<b1> ^is flat)
->
(<s1> ^stable <o1> +)
```

```
(R4-1
(goal <g1> ^problem-space <p1>)
(<p1> ^name cup)
(<g1> ^state <s1>)
(<s1> ^super-state <s2>)
(<s2> ^object <o1>)
(<s2> ^part-rel <p4>)
(<p4> ^part-of <o1>)
(<p4> ^part <c1>)
(<c1> ^isa concavity)
(<c1> ^is upward-pointing)
->
(<s1> ^open-vessel <o1> +)
```

```
(R5-1
(goal <g1> ^problem-space <p1>)
(<g1> ^super-goal <g2>)
(<p1> ^name cup)
(<g1> ^state <s1>)
(<s1> ^open-vessel <o1>)
(<s1> ^liftable <o1>)
(<s1> ^stable <o1>)
->
(<o1> ^isa cup +)
```

(a) Rules in R-chunk

```
(R1-1'
(goal <g3> ^impasse no-change)
(<g3> ^super-goal <g4>)
(goal <g4> ^state <s2>)
->
(<g3> ^problem-space <p1> +)
(<g3> ^state <s1> +)
(<p1> ^name cup +)
(<s1> ^super-state <s2> +))
```

```
(R2-1'
(goal <g3> ^problem-space <p1>)
(<p1> ^name cup)
(<g3> ^state <s1>)
(<s1> ^super-state <s2>)
(<s2> ^object <o1>)
(<s2> ^part-rel <p2>)
(<p2> ^part-of <o1>)
(<o1> ^is light)
(<p2> ^part <h1>)
(<h1> ^isa handle)
->
(<s1> ^liftable <o1> +))
```

```
(R3-1'
(goal <g3> ^problem-space <p1>)
(<p1> ^name cup)
(<g3> ^state <s1>)
(<s1> ^super-state <s2>)
(<s2> ^object <o1>)
(<s2> ^part-rel <p2>)
(<p2> ^part-of <o1>)
(<p2> ^part <f1>)
(<f1> ^isa bottom)
(<f1> ^is flat)
->
(<s1> ^stable <o1> +))
```

```
(R4-1'
(goal <g3> ^problem-space <p1>)
(<p1> ^name cup)
(<g3> ^state <s1>)
(<s1> ^super-state <s2>)
(<s2> ^object <o1>)
(<s2> ^part-rel <p2>)
(<p2> ^part-of <o1>)
(<p2> ^part <c1>)
(<c1> ^isa concavity)
(<c1> ^upward-pointing true)
->
(<s1> ^open-vessel <o1> +))
```

```
(R5-1'
(goal <g3> ^problem-space <p1>)
(<g3> ^super-goal <g4>)
(<p1> ^name cup)
(<g3> ^state <s1>)
(<s1> ^open-vessel <o1>)
(<s1> ^liftable <o1>)
(<s1> ^stable <o1>)
->
(<o1> ^isa cup +))
```

(b) Rules in I-chunk

Figure 4.5: (a) R-chunk: created by applying regression to the explanation structure (E-chunk); (b) I-chunk: created by applying the variablization to the rule traces. The structure of the R-chunk remains the same as in the E-chunk for this example.

receives the instantiations of R2 directly, as intermediate tokens, rather than receiving WMEs created from the instantiations. Thus, R1-2, R2-2, ..., R5-2 are no longer (separate) rules. To interpret (match) RU-chunks, the nonlinear Rete introduced in Chapter 3 can be used. The RU-chunk corresponds to a U-chunk.

Cost problems may be introduced in the transformation (as in chunking), because the number of instantiations of a rule can be greater than the number of WMEs created from those instantiations, as explained in [32]. For example, if object O1 has one more handle represented by two more WMEs; (Relation-1 ^part Handle-2) and (Handle-2 ^isa handle), two instantiations of R2-1 (in Figure 4.5-(a)) are created instead of one. Because these two instantiations generate the same bindings for variables <s1> and <o1>, only one tuple (WME) is generated in the problem solving. In this case, the number of tokens is increased after the WMEs are replaced with the instantiations.

4.5 Linearizing (⇒ EBL rule)

A RU-chunk can be *linearized* to become an EBL-chunk. The hierarchical structure of RU-chunks is flattened into a single layer, and the conditions are totally ordered. For example, the hierarchical structure in Figure 4.6 can be flattened into the structure in Figure 4.7. The U-chunk is also flattened to yield a chunk. After flattening, Soar/EBL and chunking use a heuristic condition-ordering algorithm to further optimize the resulting match. This linearization can increase the cost as explained in Chapter 3.

4.6 Summary

We have performed a transformational analysis of Soar/EBL. Each step has then been mapped to a corresponding transformation in chunking, and pseudo-chunks in the two systems have been compared in terms of cost and generality. These analyses and comparisons reveal that: (1) the main source of overgeneral learning in Soar stems from the need to use approximate architectural axioms, and is common to EBL and chunking; (2) the main source of overspecial learning in Soar stems from the single transformation that differs between them (chunking does instantiation-based constraining while EBL does regression); (3) chunking automatically incorporates some of Soar's architectural constraints

```
( R1-2
  (dummy)
  (dummy)
  (<g2> ^state <s2>)
```

```
(R2-2
  (R1-2)
  (R1-2)
  (R1-2)
  (R1-2)
  (<s2> ^object <o1>)
  (<s2> ^part-rel <p6>)
  (<p6> ^part-of <o1>)
  (<o1> ^is light)
  (<p6> ^part <h1>)
  (<h1> ^isa handle)
```

```
(R3-2
  (R1-2)
  (R1-2)
  (R1-2)
  (R1-2)
  (<s2> ^object <o1>)
  (<s2> ^part-rel <p8>)
  (<p8> ^part-of <o1>)
  (<p8> ^part <b1>)
  (<b1> ^isa bottom)
  (<b1> ^is flat)
```

```
(R4-2
  (R1-2)
  (R1-2)
  (R1-2)
  (R1-2)
  (<s2> ^object <o1>)
  (<s2> ^part-rel <p4>)
  (<p4> ^part-of <o1>)
  (<p4> ^part <c1>)
  (<c1> ^isa concativity)
  (<c1> ^is upward-pointing)
```

```
(R5-2
  (R1-2)
  (dummy)
  (R1-2)
  (R1-2)
  (R2-2)
  (R3-2)
  (R4-2)
```

(a) RU-chunk

```
(R1-2′
  (dummy)
  (dummy)
  (<g4> ^state <s2>)
```

```
(R2-2′
  (R1-2′)
  (R1-2′)
  (R1-2′)
  (R1-2′)
  (<s2> ^object <o1>)
  (<s2> ^part-rel <p2>)
  (<p2> ^part-of <o1>)
  (<o1> ^is light)
  (<p2> ^part <h1>)
  (<h1> ^isa handle)
```

```
(R3-2′
  (R1-2′)
  (R1-2′)
  (R1-2′)
  (R1-2′)
  (<s2> ^object <o1>)
  (<s2> ^part-rel <p2>)
  (<p2> ^part-of <o1>)
  (<p2> ^part <f1>)
  (<f1> ^isa bottom)
  (<f1> ^is flat)
```

```
(R4-2′
  (R1-2′)
  (R1-2′)
  (R1-2′)
  (R1-2′)
  (<s2> ^object <o1>)
  (<s2> ^part-rel <p2>)
  (<p2> ^part-of <o1>)
  (<p2> ^part <c1>)
  (<c1> ^isa concavity)
  (<c1> ^is upward-pointing)
```

```
(R5-2′
  (R1-2′)
  (dummy)
  (R1-2′)
  (R1-2′)
  (R2-2′)
  (R3-2′)
  (R4-2′)
```

(b) U-chunk

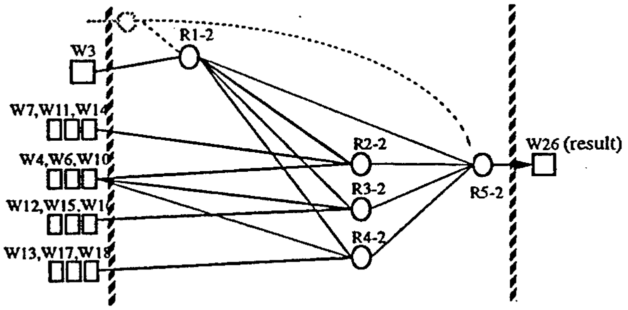


Figure 4.6: RU-chunk and U-chunk: created by eliminating intermediate rule firings in the R-chunk and I-chunk, respectively.

```
(rule EBL-rule
(goal <s2> ^state <s6>)
(<s6> ^object <o1>)
(<s6> ^part-rel <p8>)
(<p8> ^part-of <o1>)
(<o1> is light)
(<p8> ^part <h1>)
(<h1> ^isa handle)
(<s6> ^part-rel <p6>)
(<p6> ^part-of <o1>)
(<p6> ^part <b1>)
(<b1> ^isa bottom)
(<b1> ^is flat)
(<s6> ^part-rel <p4>)
(<p4> ^part-of <o1>)
(<p4> ^part <c1>)
(<c1> ^isa concavity)
(<c1> ^is upward-pointing)
-->
(<o1> ^isa cup)
```

```
(rule chunk
(goal <g4> ^state <s2>)
(<s2> ^object <o1>)
(<s2> ^part-rel <p2>)
(<p2> ^part-of <o1>)
(<o1> is light)
(<p2> ^part <h1>)
(<h1> ^isa handle)
(<p2> ^part <f1>)
(<f1> ^isa bottom)
(<f1> ^is flat)
(<p2> ^part <c1>)
(<c1> ^isa concativity)
(<c1> ^is upward-pointing)
-->
(<o1> ^isa cup)
```

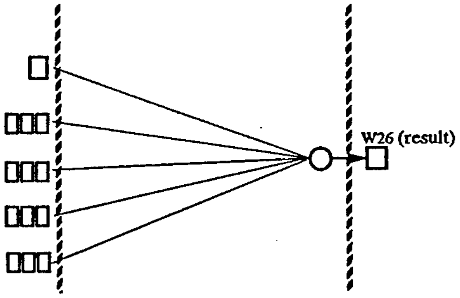


Figure 4.7: EBL-rule and chunk: results from linearizing the RU-chunk and U-chunk, respectively.

that must be added explicitly with EBL; (4) the primary sources of expensiveness in Soar's learned rules arise in three transformations that are common between chunking and EBL, and thus might have common solutions; and (5) the architectural axioms introduced to maintain tractability can cause cost change as well as overgenerality. Result (2) shows that EBL and chunking are not all that different. Result (4) goes beyond this to show that the strategies being developed to ensure that chunks are no more costly to use than was the problem solving from which they were learned, should also allow a similarly "safe" EBL mechanism to also be developed.

The following chapter describes the details of such strategies, which prevent each identified source of expensiveness (except for the cost changes by the architectural axioms), and presents a unified solution combining those strategies.

Chapter 5

Modifying the Transformations

In Chapter 3, the chunking process has been analyzed as a sequence of transformations, and intermediate products have been mapped into pseudo-chunks by providing appropriate interpreters. By evaluating the costs of the pseudo-chunks, three sources of expensiveness have been identified, and modifications that can potentially eliminate the sources have been proposed. As described in Chapter 4, a similar analysis has also been done for Soar/EBL; it has identified that chunking and Soar/EBL share the same set of sources and their solutions.

The proposed solutions convert the original sequence of transformations into a new sequence of transformations. In the new sequence, new transformations are added, and some of the original transformations are removed or modified. These alterations, however, should not violate the following key requirement: *the cost after each transformation should be bounded by the cost before the transformation.*

Figure 5.1 outlines the modifications of chunking, which keep this requirement. Chunking's original sequence of transformations, shown on the left side, is altered into a new sequence of transformations shown on the right. First, the transformation from a PS-chunk to an E-chunk is altered, since we now want to incorporate the search control in learning. One way of incorporating the search control is to simply ignore the transformation completely, and to keep all the search-control rules participating in the PS-chunk. By performing no action, instead of removing search-control rules, the cost will not increase (i.e., the cost will be bounded). However, including all of the control rules would produce excessive conditions, and sometimes make the learned rules overspecific. (Details will be provided in Section 5.2.) If the set of search-control rules overdetermine the choice, the excessive part can be pruned from the PS-chunk to make the created rule simpler. Thus, our

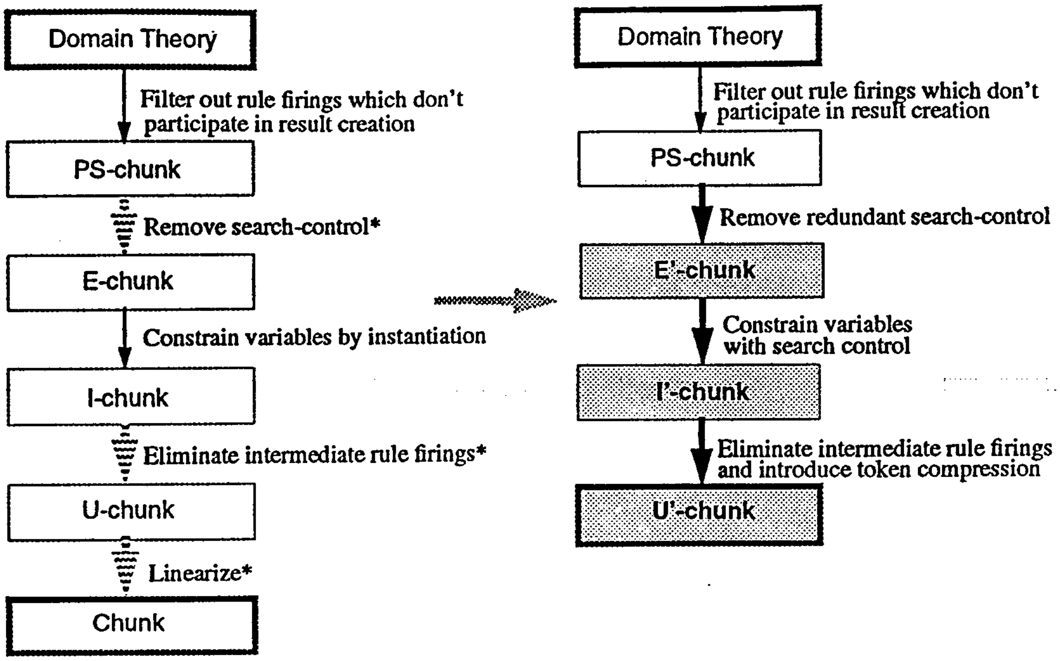

Domain Theory
Filter out rule firings which don't participate in result creation
PS-chunk
Remove search-control*
E-chunk
Constrain variables by instantiation
I-chunk
Eliminate intermediate rule firings*
U-chunk
Linearize*
Chunk
Domain Theory
Filter out rule firings which don't participate in result creation
PS-chunk
Remove redundant search-control
E'-chunk
Constrain variables with search control
I'-chunk
Eliminate intermediate rule firings and introduce token compression
U'-chunk

Figure 5.1: Outline of solutions (from the identified sources of cost increase).

new learning system introduces a new transformation that goes from a PS-chunk to a new pseudo-chunk that is called the *E′-chunk* (Extended E-chunk). An E′-chunk is generated by removing the excessive search-control rules instead of removing all the search-control rules in a PS-chunk. The interpreter for an E′-chunk includes the decision procedure, as well as the rule matcher and the rule firer.

The next step is constraining variables in an E′-chunk by examining the instantiations, as done for E-chunk to produce an I-chunk in the original transformation. The resulting pseudo-chunk is called an *I′-chunk*. An I′-chunk is different from an I-chunk in that it has additional structures originating from the search-control rules in the E′-chunk. The match process (interpretation) of an I′-chunk consists of multiple rule matches, firings, and decisions as in the case of E′-chunk.

A *U′-chunk* is generated from an I′-chunk by unifying the separate rules and decisions into a single structure. The match process of a U′-chunk is a single rule match instead of multiple rule matches, firings, and decisions. In the original transformation, to convert intermediate rule matches and firings into a single rule match, an extension of Rete (nonlinear Rete) is introduced. Because a U′-chunk has to unify decisions as well as rule matches and firings, an extension of the nonlinear Rete that can interpret the decisions is required. Also, token compression should be integrated into the interpretation. The U′-chunk is the final product in the new sequence of transformations.

This chapter discusses the above sequence of transformation in detail, including how the transformations provide relative boundedness. The first transformation from the domain theory to a PS-chunk is the same as in the original transformation. We explain how the transformation is safe (except for the changes caused by the architectural activities) in terms of the number of tokens. We then discuss; the transformation from a PS-chunk to an E′-chunk; the transformation from an E′-chunk to an I′-chunk; and the transformation from an I′-chunk to a U′-chunk, respectively. Finally, we describe similar alterations applied to Soar/EBL.

5.1 Domain Theory ⇒ PS-chunk

Before we prove that this transformation is safe, we define tools for comparing the relationships among pseudo-chunks. (In the proof, we address rule firings and decisions only, and exclude the changes caused by the architectural axioms.)

Definition 1 (trace-graph) *Given the initial WMEs (the WMEs given in the original problem solving), the trace-graph of a pseudo-chunk is a graph that represents the sequence of rule firings and decisions, along with how intermediate products are created and used in the sequence.*

In the trace-graph, each instance of a rule firing is connected to the WMEs matched to the rule conditions and the preferences produced by the rule firings. Also, each decision is connected to the input preferences and the output WMEs. The trace-graph shows the details of the rule matches, including how intermediate products are created and used to yield instantiations, depending on the match algorithm. In the case of Rete, the trace-graph shows its sharing and state saving.

Example: Figure 3.3 in Chapter 3 is an abstract view of the trace-graph of a domain theory, given the initial WMEs shown in Figure 3.2. Although the structure shows the connections among the rules and the decisions with their preferences and the WMEs, the tokens and the Rete optimizations are not shown explicitly for brevity. Figure 5.2 shows

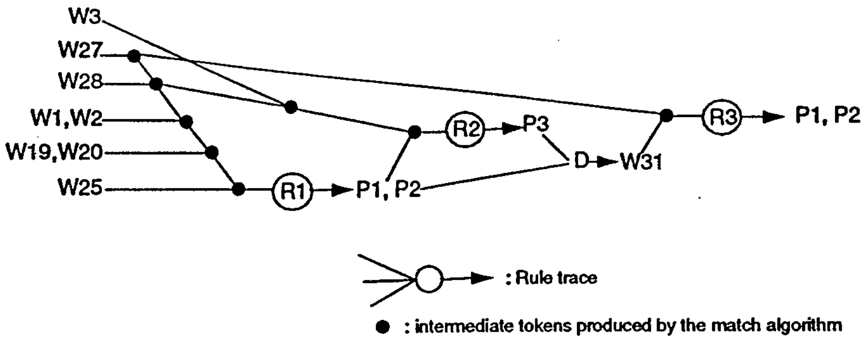


Figure 5.2: An example trace-graph.

a part of the trace-graph of the domain theory. It shows the first three rule firings. In the figure, each black bullet represents the tokens created for a condition, by *joining* the prior tokens and the WMEs matching that condition. In addition to the connections shown in Figure 3.3, it also shows how tokens are used in the match, including the sharing of tokens across the rules. For example, in the trace-graph, the tokens created for the first two conditions in R1 are shared with the tokens for the first two conditions of R2. Given a pseudo-chunk and the initial WMEs, the trace-graph of the pseudo-chunk displays the how intermediate products are created and processed in its interpretation.

Definition 2 (trace-subset) *Given the initial WMEs, a pseudo-chunk A is a trace-subset of a pseudo-chunk B, if A's trace-graph is isomorphic to a subgraph of B's trace-graph.*

Definition 3 (WME-subset). *Given the initial WMEs, a pseudo-chunk A is WME-subset of a pseudo-chunk B, if A is a trace-subset of B, and for each rule condition C in A and its mapped rule condition C′ in B, the set of WMEs matching C is a subset of the WMEs matching C′.*

Theorem 1 *Given the initial WMEs, if a pseudo-chunk A is a WME-subset of a pseudo-chunk B, the number of tokens produced while interpreting A is less than or equal to the number of tokens produced by B. That is, the number of tokens in A's trace-graph is less than or equal to that in B's trace-graph.*

proof. Because A is a trace-subset of B, each rule R in A can be mapped into a unique rule R′ in B. (Two different rules in A cannot be mapped into the same rule in B.) Also, because each condition in R matches to a subset of the WMEs matching the condition in R′ (WME-subset), there will be fewer (or the same) partial instantiations (tokens) produced while matching R than the tokens produced for R′. Thus, the total number of tokens in A's trace-graph is bounded by the total number of tokens in B's trace-graph.

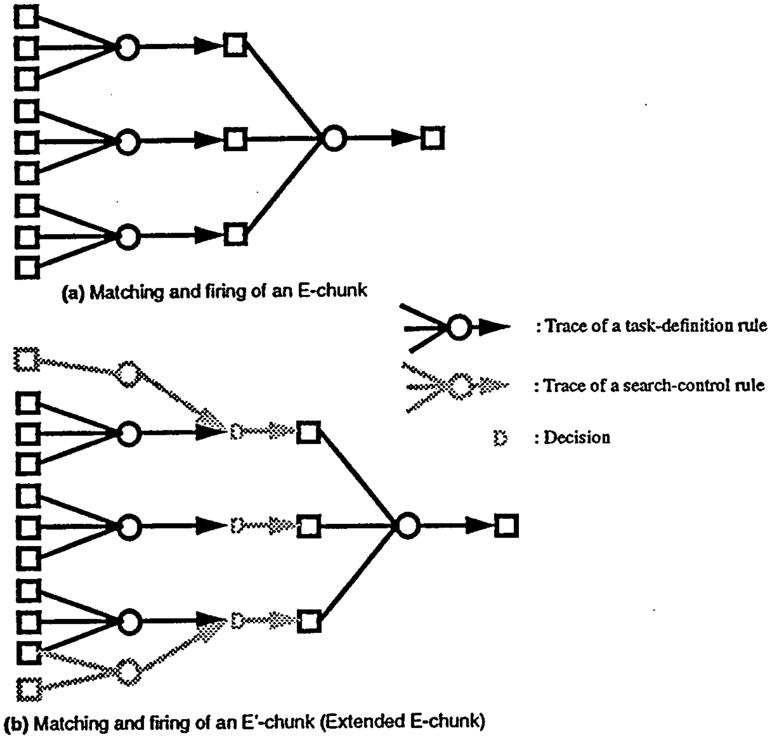
Theorem 2 *The number of tokens produced while interpreting a PS-chunk is bounded by the number of tokens produced by the problem solving episode from which the PS-chunk is created.*

proof. Because the PS-chunk is produced by eliminating the rule firings and the decisions not connected to the result creation, the problem solving episode employs either more rules and decisions than the PS-chunk's trace-graph (when there is at least one excessive rule firing), or the same rules and decisions (when there is no excessive rule firing). Thus, the PS-chunk is a trace-subset of the domain theory. Also, in the PS-chunk interpretation, only the WMEs created by the connected decision are matched by the rule condition, while in the problem solving episode, all WMEs are matched to all conditions. Thus, the PS-chunk is a WME-subset of the domain theory. By theorem 1, the number of tokens produced by the PS-chunk match is bounded by that in the problem solving episode.

5.2 PS-chunk ⇒ E′-chunk

This transformation eliminates excessive search-control rules in a PS-chunk, but incorporates (maintains) the necessary search-control rules to constrain the search in the match. The resulting E′-chunk is an extension of an E-chunk. Such an example is shown in Figure 5.3. The additional structure, colored gray, will be transformed into new conditions of the learned rule. The additional conditions can constrain the match process (match search), much as the search control does in the problem solving, and can therefore make the learned rule (and the pseudo-chunks created between this transformation and the learned rule) cheap to match.

The following subsection presents a new algorithm that determines relevant search control in decisions and removes the excessive search control. We first describe the details


(a) Matching and firing of an E-chunk

(b) Matching and firing of an E'-chunk (Extended E-chunk)

: Trace of a task-definition rule
: Trace of a search-control rule
: Decision

Figure 5.3: Extending the backtrace (explanation) to capture the search control.

Category	Preferences	Description
feasibility	Acceptable(+)	The value is a candidate for selection.
	Reject(-)	The value is not a candidate for selection.
desirability	Best(>)	The value is good enough to select without further consideration.
	Better(>), Worse(<)	Partial ordering between the candidate values.
	Worst(<)	The value should be selected only if there are no alternatives.
	Indifferent(=)	If the preference is binary (indifferent to another specific value), it means that it does not matter which one of the two is selected. If it is unary, the value is considered to be indifferent to all other values with a unary indifferent preference.
necessity	Prohibit(~)	The value cannot be selected if the goal is to be achieved.
	Require(!)	The value must be selected if the goal is to be achieved.
exclusivity	Parallel(&)	A binary parallel preference states that both values can be selected if they are not dominated by other preferences. A unary preference states that a value can be selected together with any value that also has a unary parallel preference.

Figure 5.4: Preference semantics in Soar (adapted from [34]).

of the search control semantics in Soar, then the algorithm that is built based on the semantics is presented.

5.2.1 Computing the search control

In Soar, search control is represented as preferences. Rules in Soar propose changes to working memory through preferences, each of which specifies the relative or absolute worth of a value for an attribute of a given object (as described in Section 2.2). Given a set of preferences, the decision procedure determines new WMEs based on the preference semantics. Figure 5.4 briefly describes the semantics of the major preferences.

Figure 5.5 displays the decision procedure graphically. Starting from the top, preferences are processed by nine filters. Each filter reduces the number of *candidates* (i.e., competing values) by analyzing the named preferences. Descriptions of the filters are given in Figure 5.6. Figure 5.7 provides an example. Here, there are three acceptable

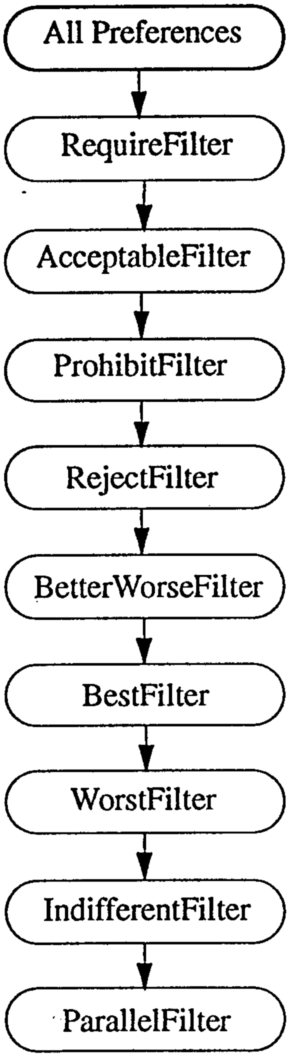


Figure 5.5: The decision procedure (adapted from [34]).

RequireFilter (!) This filter checks for required candidates and impasses.

- If there is exactly one require preference and there is no prohibit preference for the value, then its value is the winner.
- Otherwise - If there is more than one required candidate or there exists a required value that is also prohibited, recognize an impasse and exit.

AcceptableFilter (+) This filter removes the candidates thaf are not acceptable.

- If there are no acceptable preferences, then exit.

ProhibitFilter (~) This filter removes the candidates that have prohibit preferences.

RejectFilter (-) This filter removes the candidates that have reject preferences.

BetterWorseFilter (>, <) This filter checks for better/worse conflict, or filters out candidates based on better/worse preferences.

- If there are better/worse conflicts, declare an impasse and exit.
- Otherwise - It filters out the candidates that are worse than another value.

BestFilter (>) This filter removes any candidates that do not have a best preference, if there is at least one best.

WorstFilter (<) This filter removes any candidates that have a worst preference. If all the current candidates have worst preferences, the entire set is passed onto the next filter.

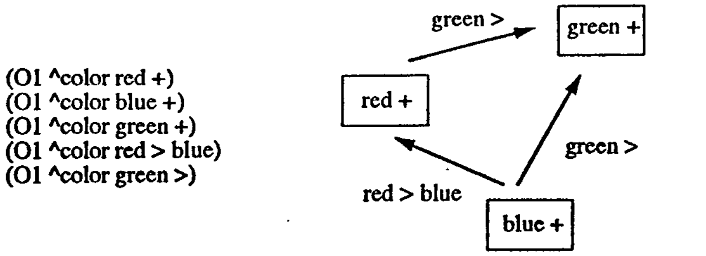
IndifferentFilter (=)

- If the candidates are all mutually indifferent, return one of the indifferent candidates.
- Otherwise - If there are non-mutually indifferent candidates for the context (goal, problem-space, state, or operator) attributes, an impasse is recognized and exit.
- Otherwise - The candidates are passed to the Parallel filter.

ParallelFilter (&)

- If all of the candidates are mutually parallel, return all of them.
- Otherwise - generate an impasse and exit.

Figure 5.6: The filter semantics (adapted from [34]).



(a) Given preferences (b) Relationship among the candidates

Figure 5.7: An example of decision.

preferences for alternative values (*red, blue* and *green*) for the same object O1 and attribute ^*color. Red* is *better* than *blue*, and *green* is *best.* In Figure 5.7-(b), each arrow shows relative strength between the two candidates. The arrowhead points to the winner and the tail points to the loser, which is determined by the preference placed beside the arrow. With these preferences, the decision procedure filters out the value *blue*, by *BetterWorseFilter*, and then the value *red* by *BestFilter.* Finally, *green* becomes the *winner* (i.e., the decided value), and WME (O1 ^*color green*) is added to the working memory.

Based on the above semantics, an algorithm has been developed to compute and collect the search-control rules at each decision point. These search-control rules would expand the explanation structure, as explained above. On first thought, it appears that the decision procedure itself might be enough to compute the search control. It might have simply collected the preferences touched during the decision. However, including all of the control rules in the explanation would produce excessive conditions, and sometimes make the learned rules overly specific. For example, in Figure 5.7, although all five preferences have contributed to the decision, two preferences, (O1 ^*color green* +) and (O1 ^*color green* >) are enough to make the same decision given the same situation. This means that they are enough to constrain the search in the match, and make the learned rule cheap. On the other hand, if all five preferences are included in the explanation, the learned rule would be less applicable than the rule derived from the explanation with the two preferences. Therefore, if the set of preferences overdetermines the choice, the redundant preferences (and their rule traces) can be pruned from the explanation to make the created rule as general as possible.

We have developed and implemented an algorithm, called the *preference collection algorithm*, to determine the relevant search control (set of preferences). The purpose of the algorithm, shown in Figure 5.8, is to find the set of relevant preferences which capture the full decision context. The algorithm makes use of the properties of each preference's semantics and the decision procedure. The input of the algorithm is the decided value (winner) as well as the preferences. Thus, the algorithm takes advantage of the fact that it already knows the winner. This allows it to skip some computations performed in the decision procedure. For instance, the algorithm does not have to consider the possibilities of another candidate being a winner. Also, because the purpose of the algorithm is different from the decision procedure and it has one more input (the winner), the algorithm examines the preferences in a different order than the decision procedure. Given the winner, the algorithm scans through the preferences in the order that yields a set of relevant preferences, where the size of the set is as small as possible (though not guaranteed to be smallest, as will be explained later). That is, it determines if a single preference can make the same decision, and if not, it then determines if two preferences can make the same decision, and so on. The set of preferences computed by the algorithm is called the *AFFECTED_PREFS* in the algorithm. Because the current explanation in Soar already includes *require, prohibit*, and the winner-proposed *acceptable* preferences, we add only extra preferences to AFFECTED_PREFS.

First, the algorithm determines if a single preference is able to yield the same decision as occurs with all given preferences. Because a *require* or sole *acceptable* preference might have decided the value without considering other preferences, they are checked in the beginning. If they fail, the algorithm examines if there are two preferences — a *best* preference for the winner and the winner-proposed *acceptable* preference. If there are, these two preferences are enough to decide the value. In this case, only the *best* preference is saved in AFFECTED_PREF because the winner-proposed *acceptable* preference is already counted in the explanation. If it also fails (i.e., there is no *best* preference for the winner), the algorithm examines other preferences one by one to exclude the remaining candidates. The algorithm first examines *prohibit* preferences, and excludes the *prohibited* candidates. Because *prohibit* preferences are already included in the explanation in Soar, these preferences are not added to AFFECTED_PREF. The algorithm then examines *reject* and *worst* preferences, if each preference can exclude one of the remaining candidates. If, while processing the *worst* preferences, it is found that the winner is one of the *worst*

Given the decided WME (*id* ^attr V), AFFECTED_PREFS is constructed. (Current explanation aleady includes require, prohibit, and winner-proposed acceptable preferences.)

```
AFFECTED_PREFS = NIL
if require-preference set is not empty exit /* require */
if there is only one acceptable preference (id ^attr V+) exit /* sole candidate */
if there is only one best preference (id ^attr V >) /* one best and one acceptable */
    insert it in AFFECTED_PREFS and exit

for all (id ^attr $v_i$ +) & ($v_i$ <> V) insert $v_i$ in CANDIDATES
for all (id ^attr $v_i$ ~) remove $v_i$ from CANDIDATES /* prohibit */

For all vi in CANDIDATES /* rejects */
    if there is (id ^attr $v_i$ -)
        insert it in AFFECTED_PREFS
        remove $v_i$ from CANDIDATES
if CANDIDATES is empty exit

TCAND = CANDIDATES /* TCAND will be used for better and worse processing */
if there is not (id ^attr V <) /* worsts ; if the winner is one of the worsts,
                                  we ignore the worst preferences*/
    for all vi in CANDIDATES
        if there is (id ^attr $v_i$ <)
            insert it in AFFECTED_PREFS
            remove $v_i$ from CANDIDATES
    if CANDIDATES is empty exit

For all $v_i$ in CANDIDATES /* better and worse */
    if there is (id ^attr $v_i$ < V) or (id ^attr V > $v_i$) /* if worse than the winner, only one
                                                  preference is required to filter the loser */
        insert one of them in AFFECTED_PREFS
        remove vi from CANDIDATES
    /* requires another preference ; worse than another acceptable value */
    else if there exist a $v_j$ such that $v_j$ is in TCAND and ((id ^attr $v_i$ < $v_j$) or (id ^attr $v_j$ > $v_i$))
        insert one of the preference and (id ^attr $v_j$ +) in AFFECTED_PREFS
        remove $v_i$ from CANDIDATES
        remove $v_j$ from TCAND
        for all $v_k$ in CANDIDATES /* remove covered elements */
            if there is (id ^attr $v_k$ < $v_j$) or (id ^attr $v_j$ > $v_k$)
                insert one of them in AFFECTED_PREFS
                remove $v_k$ from CANDIDATES

For all vi in CANDIDATES /* unary indifferent */
    if there is (id ^attr $v_i$ =), insert it in AFFECTED_PREFS
```

Figure 5.8: Preference collection algorithm.

candidates, the algorithm will ignore the *worst* preferences. Because the winner itself is one of the *worsts*, the *worst* preferences cannot filter any non-winners. This case can occur when other preceding preferences (e.g., *reject* preferences, *better* preferences, or *worse* preferences) were enough to decide the winner in the original decision (see Figure 5.6).

Better and *worse* preferences are examined last because when they are used, there sometimes needs to be two preferences to exclude one candidate. If, while examining *better* and *worse* preferences, a candidate is not directly *worse* than the decided value, the system searches for another (*acceptable*) value that is *better* than the candidate. According to the decision filters, presented in Figure 5.6, a value is not selected if it is *worse* than another *acceptable* candidate that has not been *reject*ed or *prohibit*ed. Thus, we employ a temporary set called *TCAND* that stores the non-rejected and non-prohibited candidates. If a candidate is worse than a member of TCAND, the system requires two preferences to remove the candidate — one *acceptable* and one *better* or *worse*. In this case, minimizing the number of affected preferences requires finding a minimum set of *acceptable* values, such that all the remaining candidates are *worse* than at least one of the (*acceptable*) values. However, finding a minimum set of *acceptable* preferences that is enough to cover all the remaining candidates is reducible to the set-covering problem — a known NP-complete problem. Each *acceptable* preference represents the set of candidates *worse* than the *acceptable* candidate, and the problem is to find a minimum set of *acceptable* preferences (a subset of the candidates) that is enough to cover all the remaining candidates. Our algorithm uses a simple backtrack-free heuristic to reduce the complexity to polynomial. Given a candidate, if the candidate is covered by an *acceptable* preference, the algorithm removes all the other remaining candidates covered by the *acceptable* preference. (The picked *acceptable* preference is not necessarily the best each time.) This process continues until there is no remaining candidate that is not the winner.

The computational complexity of the preference collection algorithm depends on the *better/worse* candidates processing because it requires the largest number of comparisons among the candidates to find one of the *coverings* of the remaining candidates. For each remaining candidate v_i, once it has found an element v_j, in TCAND, that is *better* than v_i, for each other remaining candidate, v_k, the algorithm examines *better/worse* preference in order to check whether v_j is also *better* than v_k. The number of *better/worse* preferences for N candidates is maximum $\mathrm{N}\times(\mathrm{N}-1)$, because for each pair of candidates, maximum two preferences — one *better* and the other *worse* — can exist. Thus, the total complexity is $O(N^4)$, when N is the number of candidates. If the system preprocess the *better/worse* preferences or employs additional indices, so that for each pair of candidates at most one preference (either a *better* or a *worse*) remains, then the total cost becomes $O(N^3)$.

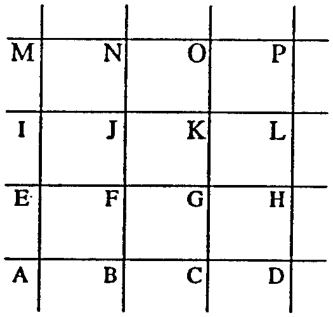
Given the preference set in Figure 5.7 and the decided value *green* by the decision procedure, the algorithm in Figure 5.8 finishes the process by the fourth line marked as '*one best and one acceptable*', and returns the preference {(O1 ^*color green* >)} as AFFECTED_PREFS. Because the winner-proposed *acceptable* preference (O1 ^*color green* +) is already included in the explanation by the current chunking process, the actual set of affected preferences consists of the two preferences: (O1 ^*color green* >) and (O1 ^*color green* +).

Since the algorithm employs a backtrack-free heuristic to reduce the computational complexity, it may produce a non-minimal set of preferences in some cases. The non-minimal set of preferences, however, is still more effective than the full set of preferences, as long as it is a subset of the other. By having fewer constraints, the learned rule is more applicable than the rule derived from the full set.

5.2.2 Decisions based on lack of knowledge

One problem with incorporating search control is that it does not specify what to do when decisions in a search are based on *lack of knowledge*. In such circumstances, the learning process has no explanation for why a choice was made, and therefore can acquire rules that are just as expensive as those learned by the unaltered learning mechanism. This problem is called the *opaque-decision problem*. In Soar, a real lack of knowledge, as reflected in an insufficient set of preferences about a decision, leads to an impasse rather than to a decision. Thus, it might seem that Soar would not suffer from this problem. However, it does have a construct — an *indifferent* preference — that allows the explicit statement of indifference among a set of choices. The decision procedure is then free to select randomly among the indifferent choices. The resulting choice (i.e., opaque decision) is thus made in such a way that no explanation of the selection among the indifferent alternatives is possible.

Consider the example from the Grid task again — the problem is to go from point F to point P, a path of length four (Figure 5.9-(a)). Because point F is adjacent to four other points, four operators are suggested, one for each direction, by the rule *operator-goto*



```
(F ^next B)
(F ^next E)
(F ^next J)
(F ^next G)
(G ^next F)
      .
      .
      .
```

(a)

```
(sp operator-goto
   (goal <g> ^problem-space <p>
             ^state <s>)
   (<p> ^name grid-path)
   (<s> ^at <loc1>)
   (<loc1> ^next <loc2>)
 -->
   (<o> ^name goto-loc
        ^from <loc1> ^to <loc2>)
   (<g> ^operator <o>))
```

(b)

Figure 5.9: The Grid task.

(Figure 5.9-(b)). If the knowledge required to choose among them is not directly available in productions (as in the case of this task), an impasse occurs on operator selection. In the subgoal created for this impasse, Soar employs the *selection* problem space, which contains evaluation operators that can be applied to the competing task operators. These evaluations, once generated, are turned into preferences that allow selection of the task operators. However, the system may lack knowledge about which one to evaluate first, and might thereby create another impasse. To avoid such impasses, Soar's background knowledge creates *indifferent* preferences for the evaluation operators. This allows it make an opaque decision (pick one randomly), and begin to make progress. Figure 5.10 shows this search process, which continues until the point P is reached.

If, as is often the case, the information about how to evaluate an operator is not directly available, an evaluation subgoal (to implement the evaluation operator) is created. The task in this third-level subgoal is to determine the utility of the operator. Here, the system performs a lookahead search trying to apply the selected task operator in the original problem space. If the resulting state can be evaluated, then the subgoal terminates; otherwise the process continues, recurring on the question of what task operator to apply to this new state.

In this overall lookahead search, indifferent preferences indirectly determine which path the system moves down, by directly determining which of the operators are evaluated

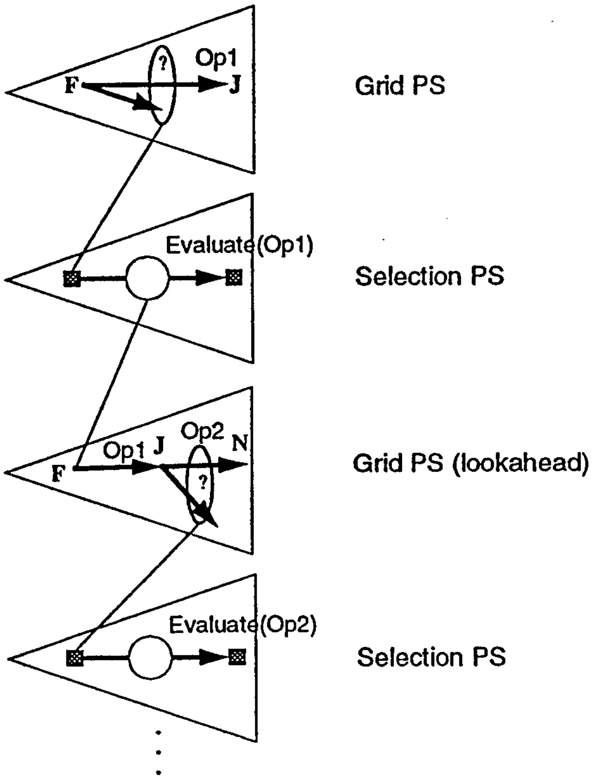


Figure 5.10: Problem solving in the Grid task.

at each point. However, the rules learned from this search cannot gather an explanation from the random semantics (indifferent preferences) as to why one path was taken rather than another. This leads to an exponential match of rules which are learned from this process. We saw an example of this in the introduction.

There are (at least) two possible ways of solving this problem. The first one is to disallow the use of indifferent preferences. Instead of selecting randomly among alternatives, an explicit default ordering is given to the alternatives. If there are any apparent reasons why one alternative should be selected ahead of another, they can be incorporated into this ordering. Where there are no such reasons, arbitrary ordering can be imposed. For the Grid task, an ordering of the operators can be assigned, according to the direction of movement. For example, an ordering of first *right*, then *up*, then *left*, and finally *down*, can be provided by the rules shown in Figure 5.11-(a). It is important to note that this ordering is just used in place of the indifferent preferences on the evaluate operators in the selection space. Thus, it determines the order in which the operators are evaluated, but does not dictate an ordering on the task operators. This latter ordering is still to be learned, as a new set of control rules, from the lookahead search.

Because the orderings are generated explicitly by the rules that distinguish among the alternatives, they leave behind a trace that can be used in explaining why one alternative is picked over the others. A key point is that it does not explain why it should be selected first, only why it actually was; so, it is descriptive rather than normative. For example, Figure 5.11-(b) shows the match search of the learned rule in the grid task. The additional constraint introduced by the conditions marking the directions and their priorities may not capture a suitable level of generality to support transfer to related situations; however, it will at least be sufficient to distinguish the one selected alternative from the others during the match, and thus be able to make the resulting learned rules (and the pseudo-chunks created between this step and the learned rule) cheap.

The second way of dealing with opaqueness caused by the randomness is to reflect this random semantics in learning and matching. In the problem solving episode, if the system has made an opaque decision, the corresponding part (conditions) of the rule that is learned from the search, can be provided with *random semantics*. Every time a condition is matched against a WME, although the condition is general enough to match any value, only one of the values is selected randomly, instead of all of them. For example, consider the learned rule from the Grid-task problem solving as shown in Figure 5.10, and the

```
(rule priority-right
  (goal <g> ^problem-space <p2>
          ^eval-operator <e1>)
  (<p2> ^name selection)
  (<e1> ^object <o1>)
  (<o1> ^name goto-loc)
          ^from <loc1> ^to <loc2>)
  (<loc1> ^right <loc2>)
  -->
  (<e1> ^priority 1))
```

```
(rule priority-up
  (goal <g> ^problem-space <p2>
          ^eval-operator <e1>)
  (<p2> ^name selection)
  (<e1> ^object <o1>)
  (<o1> ^name goto-loc)
          ^from <loc1> ^to <loc2>)
  (<loc1> ^up <loc2>)
  -->
  (<e1> ^priority 2))
```

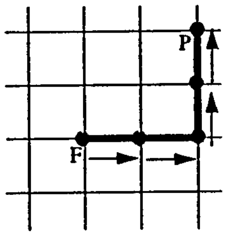
```
(rule priority-left
  (goal <g> ^problem-space <p2>
          ^eval-operator <e1>)
  (<p2> ^name selection)
  (<e1> ^object <o1>)
  (<o1> ^name goto-loc)
          ^from <loc1> ^to <loc2>)
  (<loc1> ^left <loc2>)
  -->
  (<e1> ^priority 3))
```

```
(rule priority-down
  (goal <g> ^problem-space <p2>
          ^eval-operator <e1>)
  (<p2> ^name selection)
  (<e1> ^object <o1>)
  (<o1> ^name goto-loc)
          ^from <loc1> ^to <loc2>)
  (<loc1> ^down <loc2>)
  -->
  (<e1> ^priority 4))
```

```
(rule eval-preference
  (goal <g> ^problem-space <p2>
          ^eval-operator <e1> <e2>)
  (<p2> ^name selection)
  (<e1> ^priorit <p1>)
  (<e2> ^priority <p2> { ><p1>})
  -->
  (<g> ^eval-operator <e1> > <e2>)
```

(a) Providing a default ordering among the evaluation operators

```
(...
(<r> ^priority 4 ^at <l1> ^to <l2>)
(<u> ^priority 3) (<d> ^priority 1)
(<s> ^at <l1>) (<d1> ^at <l8>)
(<l2> ^right <l3> ^next <l3>
    ^down <l4> ^next <l4>
    ^up <l5> ^next <l5>)
(<l3> ^up <l6> ^next <l6>
    ^down <l7> ^next <l7>)
(<l6> ^up <l8> ^next <l8>)
-->
(<g> ^operator <r> >))
```

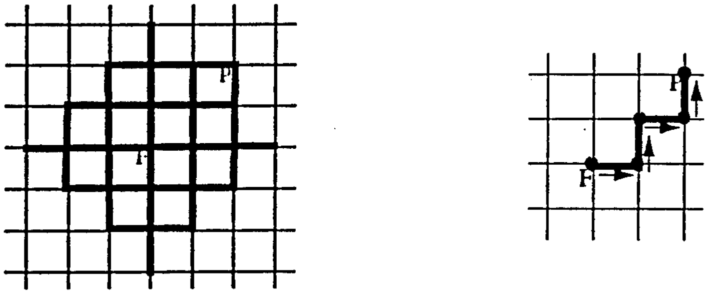


(b) Learned rule and its match search

Figure 5.11: Eliminating indifferent preferences and their opaque decisions.

```
(...
 (<state> ^at <l1>)
 (<desired> ^at <l5>)
 (<op> ^at <l1> ^to <l2>)
 (<l2> ^next <l3>)
 (<l3> ^next <l4>)
 (<l4> ^next <l5>)
 -->
 (<g> ^operator <o> >))
```

(a) Learned rule



(b) Match searh with normal match

(c) Match search with random semantics

Figure 5.12: Match search with/without random semantics.

learned rule as shown in Figure 5.12-(a). When the system matches the conditions which are built from the the opaque decisions in the problem solving, instead of matching the conditions with all WMEs, only one of them is picked randomly. For example, the match search for the rule in Figure 5.12-(a) can be changed from Figure 5.12-(b) to Figure 5.12-(c). The reflection of an opaque decision in the match leads to a single-path match. If, in fact, the indifferent preference meant that the system really didn't care which of the paths was taken, then any random selection made by the matcher should be as good as any other. If, however, the indifferent preference actually signified a lack of knowledge about the correct path, and that not all paths actually lead to success, then the match will follow one path randomly, and thus will succeed only stochastically. If, in the Grid task example, the system picks one of the wrong paths, the search would not reach the desired point.

Our implementation supports both ways of solving the problem. The first option needs only an explicit evaluation order among the alternatives, and does not demand any modification of the basic architecture of Soar, except for eliminating *indifferent* preferences. The second option requires a significant alteration in the match of the learned rule. The random semantics are applied to the match of the learned rule, and the current match algorithm (Rete) is extended to support the semantics. This extension is also combined with other extensions of the match algorithm, which are required for implementing other proposed modifications (e.g., introducing the problem-solving structure in the match). Details of such extensions, and how to combine them will be discussed in Section 5.4.

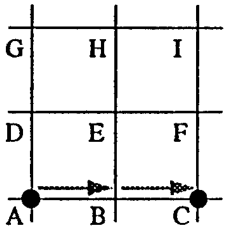
5.2.3 An example E′-chunk

The simplified Grid task described in Chapter 3 has no excessive search-control rule firings. If the Grid task is extended, by giving one more rule, R5, and a WME, W51, as shown in Figure 5.13, the problem solving episode from the new task will produce excessive search-control rule firing. The problem solving episode is shown in Figure 5.14. For the second decision in Figure 5.14, the preference (G1 ^operator A -) is excessive and can be excluded from the explanation. The excessive part is marked as thick lines in the figure. The preference collection algorithm, shown in Figure 5.8, filters the *reject* preference from the set of preferences participating in the decision. Figure 5.15 shows the extra match effort performed by the excessive part. The total cost of the problem solving episode is 29 in the new task.

The interpretation of the E′-chunk, generated from the PS-chunk, is shown in Figure 5.16. The cost of the E′-chunk is 27, which is less than the cost of the PS-chunk. In general, when the search control computing algorithm filters *any* preferences, the E′-chunk will have fewer search-control rules than the PS-chunk, and produce fewer tokens in the match. Otherwise, the number remains the same.

Theorem 3 *The number of tokens produced while interpreting an E′-chunk is bounded by the number of tokens for the PS-chunk from which it is created.*

proof. The set of the rules in the E′-chunk is either a subset of the rules in the PS-chunk (when there is at least one search-control rule) or equivalent to it (when there is no search-control rule), because of the pruned search control (trace-subset). Since the transformation only eliminates excessive search control, each decision in the E′-chunk is constrained as in the PS-chunk. Thus, each decision produces the equivalent set of WMEs that are produced by the corresponding decision in the PS-chunk. This means that each condition in the E′-chunk is matched by the same set of WMEs as the corresponding condition in the PS-chunk (WME-subset). By theorem 1, the number of tokens produced by the E′-chunk match is bounded by that in the PS-chunk match.



W1: (A ^next B)

W2: (A ^next D)

W3: (A ^right B)

W4: (B ^next A)

W5: (B ^next C)

W6: (B ^next E)

W7: (B ^right C)

W8: (C ^next F)

W9: ~~(C ^next B)~~

W10: (D ^next A)

W11: (D ^next E)

W12: (D ^next G)

W27: (G1 ^state S)

W28: (S ^at A)

W29: (G1 ^goal-point GP)

W30: (GP ^at C)

W13: (E ^next B)

W14: (E ^next F)

W15: (E ^next H)

W16: (E ^next D)

W17: (G ^next D)

W18: (G ^next H)

W19: (B ^reachable-by C1)

W20: (D ^reachable-by V1)

W21: (A ^reachable-by V2)

W22: (C ^reachable-by V2)

W23: (E ^reachable-by V2)

W24: (G ^reachable-by V2)

W25: (V1 ^name car)

W26: (V2 ^name car)

W51: (B ^left A)

(a) Given WMEs

```
(R1
(goal <g> ^state <s>)
(<s> ^at <loc1>)
(<loc1> ^next <loc2>)
(<loc2> ^reachable-by <vehicle>)
(<vehicle> ^name <n>)
-->
(<g> ^operator <loc2> +))
```

```
(R2
(goal <g> ^state <s>)
(<s> ^at <loc3>)
(<loc3> ^right <loc4>)
(<g> ^operator <loc4> +)
-->
(<g> ^operator <loc4> >))
```

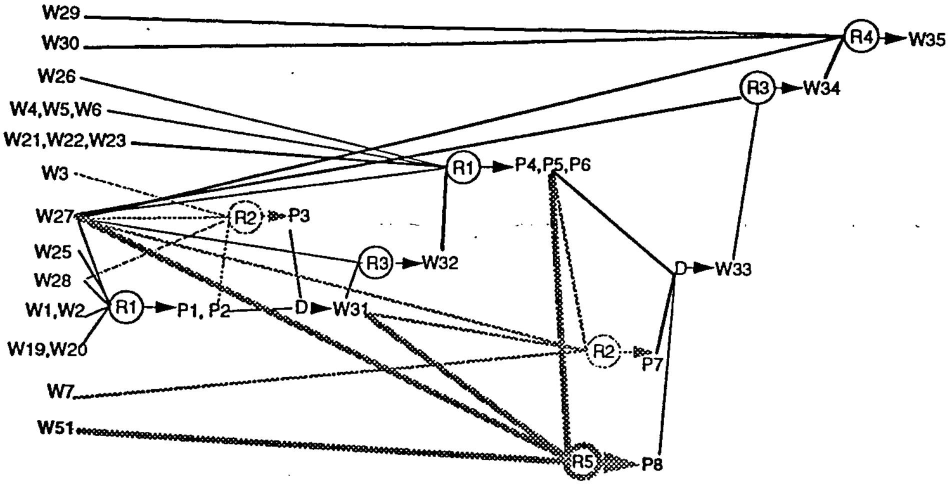
```
(R3
(goal <g> ^state <s>)
(<g> ^operator <loc5>)
-->
(<s> ^at <loc5>)
```

```
(R4
(goal <g> ^state <s>)
(<g> ^goal-point <gp>)
(<gp> ^at <loc6>)
(<s> ^at <loc6>)
-->
(<s> ^success <loc6>))
```

```
(R5
(goal <g> ^state <s>)             ; (search-control rule)
(<s> ^at <loc3>)                  ; if the current location is <loc3>,
(<loc3> ^left <loc7>)             ; and <loc4> is on the left, and
(<g> ^operator <loc7> +)          ; there is a candidate operator to goto
-->                               ; <loc4>, then reject the operator
(<g> ^operator <loc7> -))
```

(b) Given rules

Figure 5.13: An extension to the simplified Grid task.



Preferences and WMEs created during problem solving

P1: (G1 ^operator B +)
P2: (G1 ^operator D +)
P3: (G1 ^operator B >)

W31: (G1 ^operator B)
W32: (S ^at B)

P4: (G1 ^operator C +)
P5: (G1 ^operator E +)
P6: (G1 ^operator A +)
P7: (G1 ^operator C >)
P8: (G1 ^operator A -)

W33: (G1 ^operator C)
W34: (S ^at C)
W35: (S ^success C)

Figure 5.14: Problem solving episode.

```
(R5
S (goal <g> ^state <s>)                 (W27)
S (<s> ^at <loc3>)                      (W27,W32)
1 (<loc3> ^left <loc7>)                 (W27,W32,W51)
1 (<g> ^operator <loc7> CAND)           (W27,W32,.W51,P6)
-->
1 (<g> ^operator <loc7> BEST))          ==> create P7

Decision process                        P4,P5,P6,P7,P8 ==> create W33
```

Figure 5.15: Tokens created for the excessive search control.

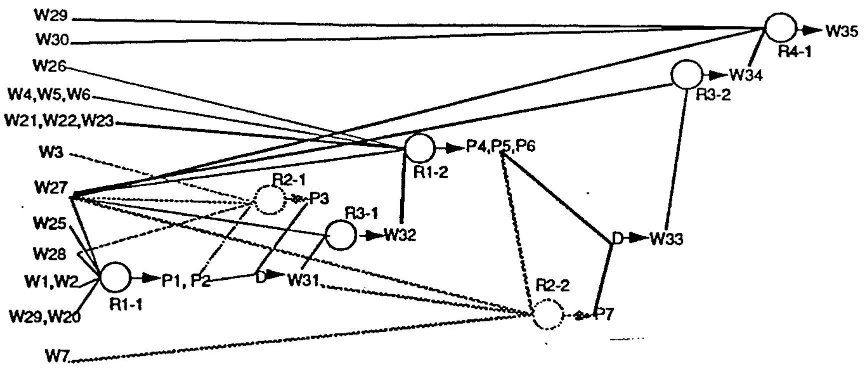


Figure 5.16: The trace of the E′-chunk.

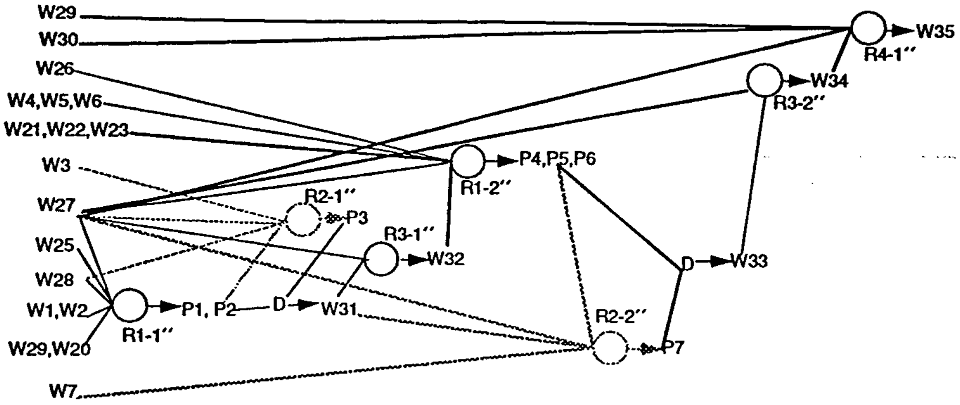
Remember that the original transformation (removal of search control) from the PS-chunk to the E-chunk has increased the cost. The cost increase is avoided now by employing the altered transformation.

The addition of search control, rather than the removal of it, may specialize the learned rules (and the pseudo-chunks created between this transformation and the learned rules), but in return, it enables the rule's cost to remain bounded by the cost of the original problem solving.

5.3 E′-chunk ⇒ I′-chunk

This step performs the variablization step in chunking by examining the rule traces in the E-chunk trace, as described in Section 3.3. This transformation can overspecialize the learned rule (and the pseudo-chunks created by the subsequent transformations) as in the original chunking's variablization process. For the Grid task example, the new I′-chunk built from the E′-chunk is shown in Figure 5.17. Variable <n1> in R1-1′ and variable <n2> in R1-2′ are overspecialized to the constant **car** as in the original transformation.

Although the characteristics of the transformation are similar to those of the original transformation, the resulting I′-chunk is different from the I-chunk in that the copies of the search-control rules and the subsequent decisions are kept in the structure. The total cost in the number of tokens is 27; the cost is the same as the match cost of the E′-chunk. In general, the number of tokens generated should be either unchanged, or reduced by the introduced constraints.



```
(R1-1"
1 (goal <g> ^state <s>)
1 (<s> ^at <loc1>)
2 (<loc1> ^next <loc2>)
2 (<loc2> ^reachable-by <v1>)
2 (<v1> ^name car)
-->
2 (<g> ^operator <loc2> ))

(R2-1"
S (goal <g> ^state <s>)
S (<s> ^at <loc1>)
1 (<loc1> ^right <loc2>)
1 (<g> ^goto-eval-operator <loc2> +)
-->
1 (<g> ^operator <loc2> >))

(R3-1"
S (goal <g> ^state <s>)
1 (<g> ^operator <loc2>)
-->
1 (<s> ^at <loc2> ))
```

```
(R1-2"
S (goal <g> ^state <s>)
1 (<s> ^at <loc2>)
3 (<loc2> ^next <loc3>)
3 (<loc3> ^reachable-by <v2>)
3 (<v2> ^name car)
-->
3 (<g> ^operator <loc3>))

(R2-2"
S (goal <g> ^state <s>)
S (<s> ^at <loc2>)
1 (<loc2> ^right <loc3>)
1 (<g> ^operator <loc3> +)
-->
1 (<g> ^operator <loc3> >))

(R3-2"
S (goal <g> ^state <s>)
1 (<g> ^operator <loc3>)
-->
1 (<s> ^at <loc3> ))

(R4-1"
S (goal <g> ^state <s>)
1 (<g> ^goal-point <gp>)
1 (<gp> ^at <loc3>)
1 (<s> ^at <loc3>)
-->
1 (<s> ^success <loc3>))
```

Figure 5.17: The interpretation of the I′-chunk that is built while learning a rule from the Grid task. An I′-chunk is created by constraining variables in an E′-chunk.

Theorem 4 *The number of tokens produced while interpreting an I'-chunk is bounded by the number of tokens of the E'-chunk from which it is created.*

proof. The I′-chunk has the same set of rules as the E′-chunk because the rules remain the same (trace-subset). The changes made by the transformation either make different variables the same or constrain the variables as constants. Because of these changes, a rule condition in the I′-chunk matches either fewer WMEs than the WMEs matching the corresponding condition in the E′-chunk (WME-subset), or the same WMEs (WME-subset). By theorem 1, the number of tokens produced by the I′-chunk match is bounded by that in the E′-chunk match.

5.4 I′-chunk ⇒ U′-chunk

This transformation has to perform two sub-transformations: (1) unifying the separate rules and decisions into one structure, and (2) applying token compression. These two sub-transformations should be applied together, because performing (1) without (2) can increase the cost, and (2) is not meaningful without (1). Token compression is needed only when intermediate WMEs are replaced by tokens, which is performed by (1).

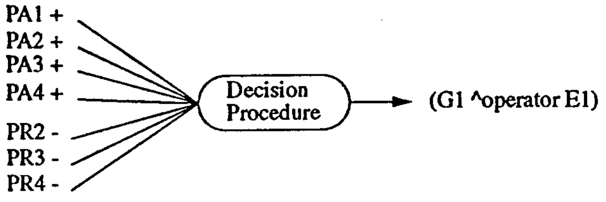
The first sub-transformation removes intermediate preferences along with the subsequent intermediate WMEs, and produces a single rule structure. In order to properly interpret this structure as a rule, an extension of the Rete algorithm is required. Nonlinear Rete, introduced to interpret U-chunk in Section 3.4, is not sufficient. Not only does the match algorithm have to interpret the hierarchical rule firing structure, but it also has to incorporate decision semantics. Token compression needs to be introduced as well, to prevent any increase in the number of tokens caused by unifying. This section introduces the extensions to nonlinear Rete that support decision semantics and token compression. First, we describe the new operations added to nonlinear Rete, so that decisions can be interpreted, and then we discuss the incorporation of token compression in extended nonlinear Rete.

```
W1: (G1 ^event E1)
W2: (G1 ^event E2)
W3: (G1 ^event E3)
W4: (G1 ^event E4)
W5: (G1 ^event E5)
W6: (G1 ^new-event E1)
W7: (G1 ^new-event E2)     (rule-acceptable            (rule-reject
W8: (G1 ^new-event E3)       (<g> ^new-event <o>)        (<g> ^event <o>)
W9: (G1 ^new-event E4)       -->                         (<o> ^has bad-result)
W10: (E2 ^has bad-result)    (<g> ^operator <o> +))      -->
W11: (E3 ^has bad-result)                                (<g> ^operator <o> -))
W12: (E4 ^has bad-result)
```

(a) WMEs and search-control rules

```
Instantiations of rule-acceptable :
acc-1: {(G1 ^new-event E1)}                         --> PA1: (G1 ^operator E1 +)
acc-2: {(G1 ^new-event E2)}                         --> PA2: (G1 ^operator E3 +)
acc-3: {(G1 ^new-event E3)}                         --> PA3: (G1 ^operator E4 +)
acc-4: {(G1 ^new-event E4)}                         --> PA4: (G1 ^operator E2 +)

Instantiations of rule-reject :
rej-1: {(G1 ^event E2) (E2 ^has bad-result)}        --> PR1: (G1 ^operator E2 -)
rej-2: {(G1 ^event E3) (E3 ^has bad-result)}        --> PR2: (G1 ^operator E3 -)
rej-3: {(G1 ^event E4) (E4 ^has bad-result)}        --> PR3: (G1 ^operator E4 -)
```

(b) Instantiations and preferences created by the instantiations



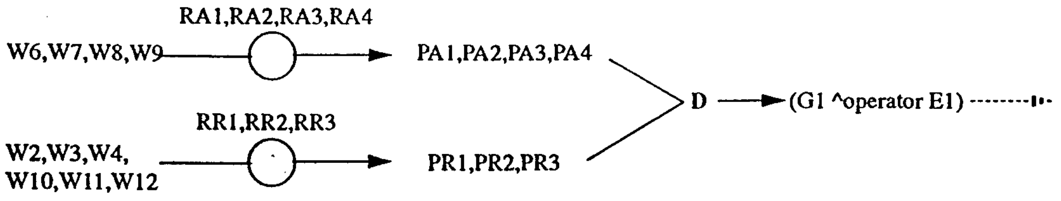
(c) Decision procedure produces a winner

Figure 5.18: An example decision in a problem solving episode.

5.4.1 Removing intermediate preferences and WMEs

Removing intermediate preferences means that the instantiations of the rules that created the preferences are directly used in the decisions (instead of creating the preferences and processing them in the decisions). This requires a new decision algorithm that embodies decision semantics to process instantiations instead of preferences. *Removing intermediate WMEs* also means that the decision algorithm does not create WMEs. The set of instantiations that participated in the decision is directly used for further matches.

Figure 5.18 shows such a decision. Given the WMEs and the rules in Figure 5.18-(a), seven preferences are created for the same id **(G1)** and attribute **(operator)**, as shown in Figure 5.18-(b). These preferences are processed by the decision procedure, as explained in Section 5.2. A WME is created as a result, as shown in Figure 5.18-(c). After the



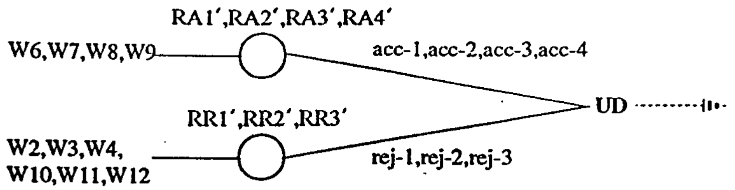
D: decision procedure call

```
(RA1                    (RA2                    (RA3                    (RA4
 (<g> ^new-event <e1>)   (<g> ^new-event <e2>)   (<g> ^new-event <e3>)   (<g> ^new-event <e4>)
 -->                     -->                     -->                     -->
 (<g> ^operator <e1> +)) (<g> ^operator <e2> +)) (<g> ^operator <e3> +)) (<g> ^operator <e4> +))

(RR1                    (RR2                    (RR3
 (<g> ^event <e2>)       (<g> ^event <e3>)       (<g> ^event <e4>)
 (<e2> ^has bad-result)  (<e3> ^has bad-result)  (<e4> ^has bad-result)
 -->                     -->                     -->
 (<g> ^operator <e2> -)) (<g> ^operator <e3> -)) (<g> ^operator <e4> -))
```

Figure 5.19: Interpretation of the rules and the decision in I′-chunk.

first three transformations (Domain Theory ⇒ PS-chunk, PS-chunk ⇒ E′-chunk, and E′-chunk ⇒ I′-chunk), the rule firings and the decisions in the problem solving episode are transformed into the rule firings and the decision shown in Figure 5.19. As explained in Section 3.3, in order to build a PS-chunk (and its subsequent E′-chunk and I′-chunk), a copy of a rule is created for each rule firing, as if the explanation structure in EBL is built by creating a separate rule copy for each instantiation. For example, four copies of rule-acceptable — RA1, RA2, RA3, and RA4 — are created for the four instantiations of the rule. In their interpretation, RA1, . . . , RA4 share one rule structure because they have the same input and output, and the same pattern of constant and variable tests. (They share one node in the figure.) In the rule level they are separate, but in the Rete level (in the compiled structure) they share one network. The letter **D** represents a procedure call of the same decision procedure employed in the problem solving. The decision in the I′-chunk is identical to the original decision, except that the WMEs created by the decision in the I′-chunk are matched only by the rules that are connected in the structure, as opposed to being matched to any conditions in the production system.

Figure 5.20-(a) shows the corresponding subrules and the decision in the U′-chunk. The intermediate preferences and WMEs are removed, and instantiations are directly processed by the decision. Subrules RA1′, . . . , RA4′ are formed from RA1, . . . , RA4, and



UD: unified decision for RA1',RA2',RA3',RA4',RR1',RR2',RR3',

```
(RA1'                       (RA2'                       (RA3'                       (RA4'
  (<g> ^new-event <e1>)       (<g> ^new-event <e2>)       (<g> ^new-event <e3>)       (<g> ^new-event <e4>)
(RR1'                       (RR2'                       (RR3'
  (<g> ^event <e2>)           (<g> ^event <e3>)           (<g> ^event <e4>)
  (<e2> ^has bad-result)      (<e3> ^has bad-result)      (<e4> ^has bad-result)
```

(a) Interpretation of the subrules and the unified decision in the U'-chunk

Input:
Instantiaitons of RA1' — RA4' and RA1' — RA4'
(acc-1 — acc-4, rej-1 — rej-3)
Output:
Instantiations of RA1'

(b) Input and output of the unified decision

Figure 5.20: Interpreting the unified decision in the U'-chunk.

can also share the same structure in their interpretation, since they have the same patterns of tests.

Since the decision in the U'-chunk is different from the normal decision procedure, in that it has to process instantiations instead of preferences, the decision is represented as **UD** (*Unified Decision*) instead of **D** to distinguish the difference. Unified decisions also differ from the normal decisions, in that they cannot be performed by calling a general decision procedure. As each rule has patterns in its conditions and actions, each unified decision has patterns for its input and output. These input and output patterns are determined by the subrules which participate in the decision. For example, as shown in Figure 5.20-(b), the given unified decision will process the instantiations of the subrule RA1', ..., RA4', and RR', ..., RR3', and *decide* the instantiation that created the winner. That is, it has to decide which instantiation is *consistent* with RA1' (the subrule that is created from the winner-proposed instantiation — acc-1), as explained later.

Figure 5.21 shows the details of the decision performed in the I'-chunk match. The decision procedure filters the candidates one by one, based on the *reject*ed values, by

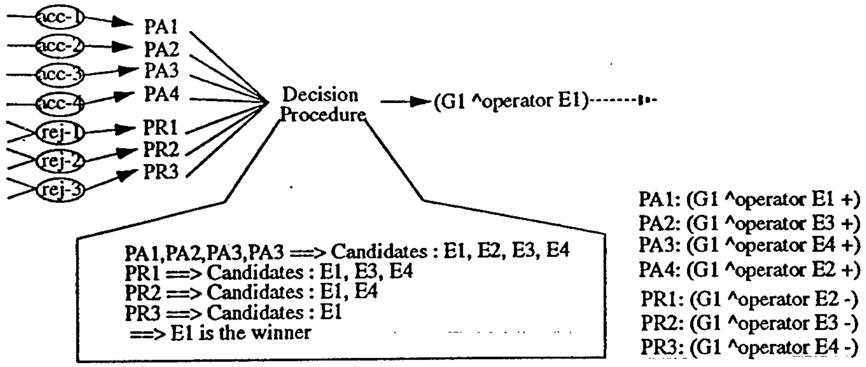


Figure 5.21: The details of the decision in the I-chunk match.

marking the *reject*ed values among the candidate values. The total cost is linear in the number of candidates. The unified decision formed from the decision should provide similar semantics to determine the winner (or the winner-proposed instantiation).

In Figure 5.22, a search that can be performed by the unified decision is shown. In this search, one reject-preference-created instantiation (we call it as *rej-instantiation* for brevity) is picked for each *reject*-preference created subrule (called the *rej-subrule*). Moreover, whenever an instantiation is picked for a rej-subrule, the system decides its consistent acceptable-preference-created instantiation (called the *acc-instantiation*).[1] This is similar to the process in the decision in the I′-chunk, which filters the candidates that are *reject*ed and find the winner. That is, the **pick one** process in the unified decision filters the three acc-instantiations that are consistent with the rej-instantiations, and finds the instantiation of RA1′. The complexity of this search is linear in the number of candidates. This process is more specialized than the original decision procedure, because its algorithm depends on the patterns of tests in the subrules, and the number of subrules in the decision. However, this guarantees the correctness and the linear time boundedness, given a similar situation. No matter which rej-instantiation is picked each time, the acc-instantiations that are consistent with the rej-instantiations will be filtered, and the system will always be left with the non-rejected candidate, and can make the correct decision.

[1] An *acceptable*-preference created subrules (called an *acc-subrule*) is *consistent* with a rej-subrule, when their corresponding rules in the I′-chunk have proposed and rejected the same candidate. Also, an acc-instantiation is *consistent* with a rej-instantiation, when they are instantiations of two consistent subrules.

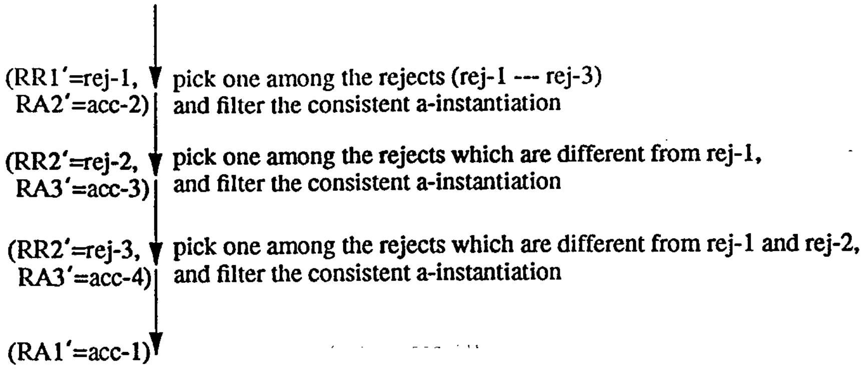

(RR1′=rej-1, RA2′=acc-2)
pick one among the rejects (rej-1 --- rej-3) and filter the consistent a-instantiation
(RR2′=rej-2, RA3′=acc-3)
pick one among the rejects which are different from rej-1, and filter the consistent a-instantiation
(RR2′=rej-3, RA3′=acc-4)
pick one among the rejects which are different from rej-1 and rej-2, and filter the consistent a-instantiation
(RA1′=acc-1)

Figure 5.22: A cheaper search for the unified decision.

Matching (interpreting) a unified structure (a U′-chunk) is a single rule match process, and the above search (a sub-part of a U-chunk match) should be performed in the rule match algorithm. This requires a significant modification in the match process. In the nonlinear Rete, introduced in Section 3.4, the only operation between two instantiations of subrules is *join*, which tests the consistency between the instantiations based on the variable patterns of the subrules. The above task, however, requires an extra operation that can pick one instantiation that is not the same as the instantiations of the preceding subrules. (The structure in Rete will be shown shortly.) For example, the above decision requires picking one instantiation among the rej-instantiations, which is different from the instantiations of the preceding rej-subrules. We have introduced an extra not type, called a *decision-sub-node*, for this purpose. A decision-sub-node picks one of the instantiations of a subrule arbitrarily, instead of keeping all consistent instantiations. This 'pick one' operation filters out *reject*ed candidates one at a time, as the decision procedure filters one *reject*ed candidate per preference. This extension of nonlinear Rete is called *controlled-nonlinear-Rete*. The search, shown in Figure 5.22, can be performed by controlled-nonlinear-Rete, as shown in Figure 5.23. (The structure in the figure does not reflect the sharing optimization of Rete.) Each parenthesized subrule name in the figure represents the Rete sub-network that interprets that subrule. Each acc-subrule network is paired with its consistent rej-subrule network via a *join* node to filter the candidate (acc-instantiation) that is consistent with the rej-instantiation. Each pair is interpreted as a subrule (called a *decision-sub-condition*) in the network. The decision-sub-nodes are marked as DN. Each DN compares

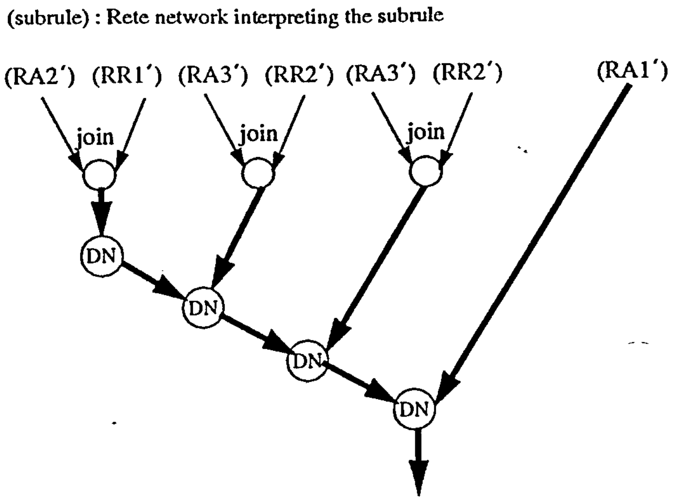


Figure 5.23: The controlled-nonlinear-Rete network built for the unified decision.

the acc-instantiations of the current decision-sub-condition with the acc-instantiations of the preceding decision-sub-conditions. The last DN finds the instantiation of RA1′.

The actual Rete network with the sharing optimization is shown in Figure 5.24. Since the acc-subrules have the same pattern of tests, they can share one network marked as (acc-subrule) — the network interpreting an acc-subrule. Also, the rej-subrules share one network marked as (rej-subrule). The decision-sub-conditions, created by pairing a rej-subrule and an acc-subrule, can share one sub-network. This network performs the process described above.

Since the decisions are interpreted as additional Rete nodes in the controlled nonlinear Rete, the cost of the decision procedure (originally performed by the Soar architecture) is converted into a sub-part of the match cost. However, because each decision-sub-node keeps at most one token, the increase in number of tokens is linear in the number of candidates. Thus, it has the same complexity as the original decision procedure, and the actual total cost does not increase.

Note that the trace-graph of the U′-chunk reflects the interpretation by controlled-nonlinear-Rete, and displays the structure shown in Figure 5.24.

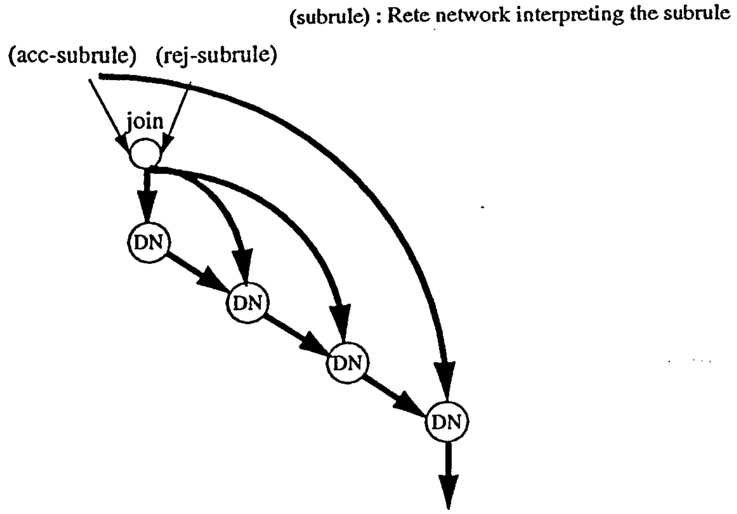


Figure 5.24: The same controlled-nonlinear-Rete showing the sharing.

5.4.2 A general unified decision structure

Figure 5.25 shows the general structure of a unified decision. Each symbol enclosed in braces represents a search-control-created subrule, such as acc-subrule or rej-subrule. The parts marked as DS represent the decision-sub-conditions. Each decision-sub-condition is interpreted as a decision-sub-node in controlled-nonlinear-Rete, as described in the last subsection. Given a set of search-control-created subrules, a list of decision-sub-conditions are created, and are interpreted by controlled-nonlinear-Rete. For example, given the set of subrules, RA1′, . . . , RA4′ and RR1′, . . . , RR3′, a list of decision-sub-conditions can be created, as shown in Figure 5.26. These conditions are interpreted as the network shown in Figure 5.24.

The general structure of a unified decision follows the decision procedure semantics, and is consistent with the sequence of filters shown in Figure 5.5. If there is a *require* preference or only one *acceptable* preference in a decision, only one subrule is created for the unified decision, and this trivial decision can be treated as a simple subrule, without an decision-sub-condition. The tokens of other preference-created subrules are processed similarly, as in the earlier example (in the previous subsection), except for the tokens of the *best*-preference-created subrules. A *best* preference, if it has participated in filtering candidates in the original decision procedure (and picked up by the preference collection algorithm shown in Figure 5.8), should be consistent with the winner. Thus,

```
        DS{
          {acceptable-preference-created-subrule-p-1}
          {prohibit-preference-created-subrule-1}
        }
                              .
                              .
        DS{                   .
          {acceptable-preference-created-subrule-p-n}
          {prohibit-preference-created-subrule-n}
        }
        DS{
          {acceptable-preference-created-subrule-r-1}
          {reject-preference-created-subrule-1}
        }
                              .
                              .
        DS{                   .
          {acceptable-preference-created-subrule-r-n}
          {reject-preference-created-subrule-n}
        }
        DS{
          {acceptable-preference-created-subrule-b-1-1}
          {acceptable-preference-created-subrule-w-1-2}
          {better-or-worse-preference-created-subrule-1}
        }
                              .
                              .
        DS{                   .
          {acceptable-preference-created-subrule-b-n-1}
          {acceptable-preference-created-subrule-w-n-2}
          {better-or-worse-preference-created-subrule-n}
        }
        DS{
          {acceptable-preference-created-subrule-w-1}
          {worst-preference-created-subrule-1}
        }
                              .
                              .
        DS{                   .
          {acceptable-preference-created-subrule-w-n}
          {worst-preference-created-subrule-n}
        }
        DS{
          {acceptable-preference-created-subrule-i-1}
          {indifferent-preference-created-subrule-1}
        }
                              .
                              .
        DS{                   .
          {acceptable-preference-created-subrule-i-n}
          {indifferent-preference-created-subrule-n}
        }
        DS{
          {acceptable-preference-created-subrule-i-1}
          {parallel-preference-created-subrule-1}
        }
                              .
                              .
        DS{                   .
          {acceptable-preference-created-subrule-pl-n}
          {parallel-preference-created-subrule-n}
        }
DS{
  {acceptable-preference-created-subrule-b}      OR      DS{acceptable-preference-created-subrule}
  {best-preference-created-subrule}
}
```

Figure 5.25: The general structure of a unified decision.

```
DS{
  {(<g> ^new-event <e2>)}
  {(<g> ^event <e2>)
   (<e2> ^has bad-result)}
}
DS{
  {(<g> ^new-event <e2>)}
  {(<g> ^event <e2>)
   (<e2> ^has bad-result)}
}
DS{
  {(<g> ^new-event <e2>)}
  {(<g> ^event <e2>)
   (<e2> ^has bad-result)}
}
{(<g> ^new-event <e1>)}
```

Figure 5.26: The structure of the example unified decision.

the *best*-preference-created subrule is paired with the winner-created subrule, as shown at the bottom of the list of subrules. For *better* or *worse* preference created subrules, two acc-subrules (one for the *better* and the other for the *worse*) are put together, since the two candidates need to be compared as in the original decision procedure.

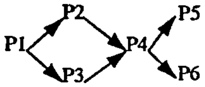
The overall interpretation of the general structure is as follows. First, any *prohibit*-created or *reject*-created subrule pick one instantiation. Next, for each picked instantiation, the consistent acc-instantiation is compared with the already filtered acc-instantiations, to make sure that it is not already filtered by other subrules. Then *better/worse*-created subrules filter the candidates. In the original decision procedure, *better/worse* preferences compare the candidates that are not already filtered by *reject* or *prohibit* preferences. Since the structure keeps the two acc-subrules that are consistent with the two candidates for each *better/worse*-created subrule, by comparing the acc-instantiations (consistent with the picked *better/worse*-created instantiation) with already *prohibit*ed or *reject*ed acc-instantiations, the same result can be acquired. It then filters out the *worst* instantiations. If there is not a *best*-preference-created subrule, the acc-instantiation which is not consistent with the already filtered acc-instantiations becomes the winner. Otherwise, the acc-instantiation which is consistent with the instantiation of the *best*-preference created subrule (and different from the filtered acc-instantiations) becomes the winner.

Note that the random semantics of the *indifferent* preferences (opaque decision) discussed in subsection 4.1.2 can be easily captured in this framework. Each *indifferent*-preference-created subrule, except for the one consistent with the winner, can be paired with the consistent acc-subrule to create a decision-sub-condition. Each decision-sub-condition is interpreted by a decision-sub-node, and the decision-sub-nodes will pick one of the candidates (acc-instantiations) for it. The candidate that is not picked by these nodes becomes the winner. For example, if there were four options in the original problem solving, the unified decision will distribute the three options into one of the three *indifferent*-preference-created subrules, and the candidate, which happened to not be picked by them, becomes the winner.

5.4.3 Applying token compression

Without token compression, all the instantiations of the sub-rules are directly transferred to the connected rules in the U′-chunk match. Figure 5.27 shows an example. Given the rules and WMEs in Figure 5.27-(a), matching rule Rule-next creates four instantiations: I1, ..., I4, as shown in Figure 5.27-(b). Because all of these instantiations have the same values for the variables in the action (<state> and <point3>) in the I′-chunk, only one WME (W1) is created from them in the I′-chunk match. The new WME W1 is matched to the first condition of Rule-connected′. Figure 5.27-(c) shows a part of the U′-chunk match built without token compression. Because the four instantiations are directly passed to Rule-connected″, the match cost can increase. At least for the first condition of Rule-connected″, the number of tokens increases from 1 to 4.

To avoid the above increase in the number of tokens, token compression merges the equivalent tokens into one token. Because the variables in the action determine the unique WMEs created by the action execution, one way of implementing token compression is to explicitly represent only the values of the variables in the action. We call the variables in the action *exposed variables.* Thus, the exposed variables for the action of Rule-next are <state> and <point3>. (The algorithm that computes the exposed variables is given later.) Given these exposed variables, as shown in Figure 5.28-(a), the four instantiations are merged into one tuple (State P4), and this tuple is used instead of the four instantiations in the U′-chunk match. Because the tuple represents any of the four instantiations, it is not removed until all of the four instantiations are removed. Figure 5.28-(b) shows a part of



```
(State1 ^at P1)
(P1 ^next P2) (P1 ^next P3)
(P2 ^next P4) (P3 ^next P4)
(P4 ^next P5) (P4 ^next P6)
```

```
(Rule-next
 (<state> ^at <point1>)
 (<point1> ^next <point2>)
 (<point2> ^next <point3>)
 (<point3> ^next <point4>)
 -->
 (<state> ^connected-to-2 <point3>))
```

```
(Rule-connected
 (<state> ^connected-to-2 <point3>)
        :
 -->
        :
```

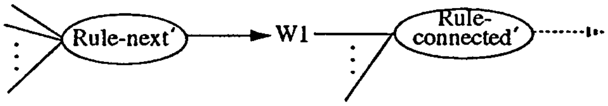
(a) Given WMEs and rules

Instantiations of Rule-next:

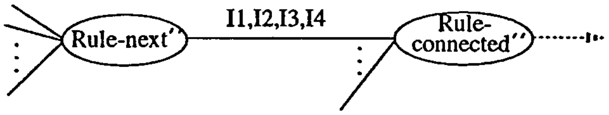
I1: (State ^at P1) (P1 ^next P2) (P2 ^next P4) (P7 ^next P5)
I2: (State ^at P1) (P1 ^next P2) (P2 ^next P4) (P4 ^next P6)
I3: (State ^at P1) (P1 ^next P3) (P3 ^next P4) (P4 ^next P5)
I4: (State ^at P1) (P1 ^next P3) (P3 ^next P4) (P4 ^next P6)

Newly created WMEs

W1: (State ^connected-to-2 P4)

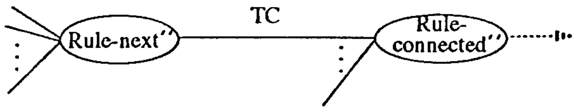


(b) I′-chunk match



(c) U′-chunk match without token compression

Figure 5.27: Building a U′-chunk without token compression.



```
(<state> ^at <point1>)
(<point1> ^next <point2>)      TC
(<point2> ^next <point3>)  ___________  C1(<state> ^connected-to-2 <point3>)
(<point3> ^next <point4t>)
        TC (based on <state> <point3>) : I1, I2, I3, I4 are all equivalent
                                         ==> create (State P4) - (I1,I2,I3,I4)
```

(a) U′-chunk match with token compression

```
(Rule-connected''
  Rule-next'' (<state> <point3>)
   {
    (<state> ^at <point1>)
    (<point1> ^next <point2>)
    (<point2> ^next <point3>)
    (<point3> ^next <point4>)
   }
             :
```

(b) U′-chunk with token compression

Figure 5.28: Building a U′-chunk with token compression.

U′-chunk created from the I′-chunk. The U′-chunk is one rule created by unifying the two rules, Rule-connected′ and Rule-next′. The name of the subrule in the U′-chunk is Rule-connected″, and its first condition is a nonlinear condition that tests subrule Rule-next″. The exposed variables of the nonlinear condition are in bold-face.

Unifying rules replaces intermediate WMEs with instantiations that created the WMEs, and the application of token compression to the unified structure replaces the instantiations with the tuples of the exposed variables' values. The tuple is different from a WME, in that its creation and deletion are performed within one rule (U′-chunk) match in controlled-nonlinear-Rete, instead of multiple rule matches and decisions. In general, because the number of the tuples is always bounded by the number of WMEs and the tuples provide the same information about the bindings of the exposed variables as the WMEs, a cost increase (increase in the number of tokens) by unifying can be avoided, though a constant overhead is added. That is, the total number of tokens is bounded by the number of tokens in the I′-chunk match, instead of increasing. The proof will be given in subsection 5.4.5.

When one rule has multiple actions, the exposed variables for the different actions can be different. For example, given the new rules in Figure 5.29-(a), the firing of the rule Rule-next-2 creates WMEs based on the two actions, and the new WMEs are matched to the two conditions of the rule Rule-connected-2, as shown in Figure 5.29-(b). In the I′-chunk match, the four instantiations of Rule-next-2′ create one WME (W1) by executing the first action, and W1 is matched to Rule-connected-2′'s first condition. Also, the four instantiations create two WMEs (W2 and W3) by executing the second action, and they are matched to the second condition of Rule-connected-2′.

Figure 5.29-(c) describes the token compression process for these rules. Based on the exposed variables of the two actions, two different sets of tuples are created. For the first action, the exposed variables are <state> and <point3>. For these variables, all of the instantiations are equivalent and only one tuple is created as the token, and it matches to the first condition of Rule-connected-2″, as marked as TC1 in Figure 5.29-(c). For the second action, the exposed variables are <state> and <point4>. For these two variables, two tokens are created and matched to Rule-connected-2″'s second condition. Figure 5.29-(d) shows the U′-chunk created from the I′-chunk. It has the same subrule (Rule-connected-2″) for the first two (nonlinear) conditions. The second condition is marked **SHARED** in the figure. Although they share one subrule, they are different in the use of their instantiations. Depending on the exposed variables, different tokens are created as the instantiations of the subrule. The next subsection describe the algorithm that computes exposed variables.

5.4.4 Computing exposed variables

For token compression, the system computes exposed variables for each variable in the actions. If a given variable is in the LHS, we simply use the variable. Figure 5.30 shows a part of an I′-chunk. In the figure, each arrow that links an action to a condition represents that the WME created by the action is matched by the condition. In the case of Rule-1, the exposed variable for the action are simply <p3> and <s>.

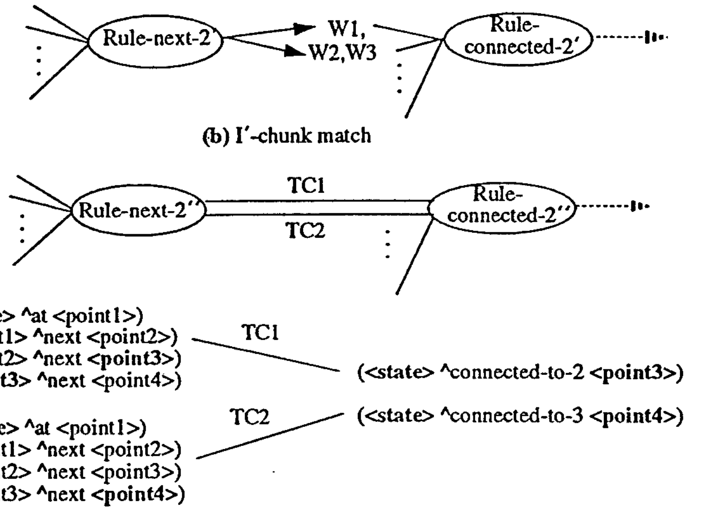
If a variable is *new*, in that it is not in the LHS, the execution of the action creates a new object. In Soar, for instance, non-operational (non-supergoal) objects, such as new operators in a subgoal, can be created in the subgoal and tested while problem solving. All non-operational variables in Figure 5.30 are marked in bold-face. Because only the operational variables are directly accessible, to compute the exposed variables for a non-operational variable, the system has to find the set of operational variables that uniquely determine the variable. For example, in Rule-2, the non-operational variable

```
(Rule-next-2                                  (Rule-connected-2
 (<state> ^at <point1>)                       (<state> ^connected-to-2 <point3>)
 (<point1> ^next <point2>)                    (<state> ^connected-to-3 <point4>)
 (<point2> ^next <point3>)                    -->
 (<point3> ^next <point4>)                    ...
 -->
 (<state> ^connected-to-2 <point3>))
 (<state> ^connected-to-3 <point4>))
```

(a) New rules

Newly created WMEs

W1: (State ^connected-to-2 P4)
W2: (State ^connected-to-3 P5)
W3: (State ^connected-to-3 P6)



(b) I'-chunk match

```
(<state> ^at <point1>)
(<point1> ^next <point2>)          TC1
(<point2> ^next <point3>)
(<point3> ^next <point4>)                    (<state> ^connected-to-2 <point3>)

(<state> ^at <point1>)             TC2       (<state> ^connected-to-3 <point4>)
(<point1> ^next <point2>)
(<point2> ^next <point3>)
(<point3> ^next <point4>)
```

TC1 (based on <state> <point3>) : I1, I2, I3, I4 are all equivalent
--> create (State P4)

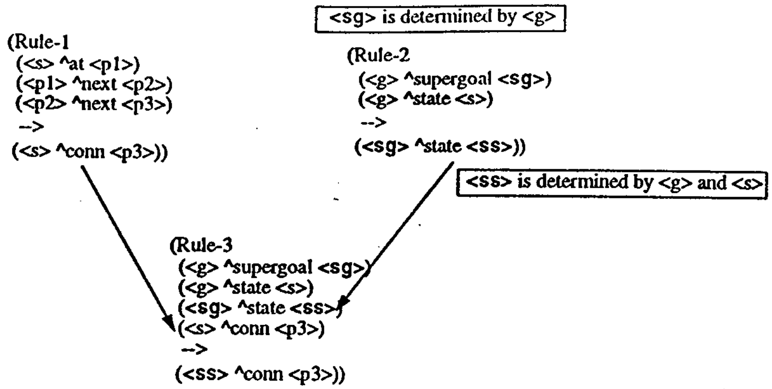
TC2 (based on <state> <point4>) : I1 and I3 are equivalent, I2 and I4 are equivalent
--> create (State P5) (State P6)

(c) U'-chunk match with token compression

```
(Rule-connected-2''
  Rule-next-2'' (<state> <point3>)
   {
    (<state> ^at <point1>)
    (<point1> ^next <point2>)
    (<point2> ^next <point3>)
    (<point3> ^next <point4>)
   }

  SHARED Rule-next-2'' (<state> <point4>)
          :
```

(d) U'-chunk with token compression

Figure 5.29: Token compression with multiple actions.



```
(Rule-1
 (<s> ^at <p1>)
 (<p1> ^next <p2>)
 (<p2> ^next <p3>)
 -->
 (<s> ^conn <p3>))

(Rule-2
 (<g> ^supergoal <sg>)
 (<g> ^state <s>)
 -->
 (<sg> ^state <ss>))

(Rule-3
 (<g> ^supergoal <sg>)
 (<g> ^state <s>)
 (<sg> ^state <ss>)
 (<s> ^conn <p3>)
 -->
 (<ss> ^conn <p3>))
```

Figure 5.30: The hierarchical condition structure of the I′-chunk.

```
compute_exposed_variables (var) {
    if var is a member of the aleady marked non-operational variables,
        return the exposed variables for var ;
        /* The computation requires O(total number of non-operational variables) time */
    elseif var is a member of the variables in the conditions,
        return the var
        /* The computation requires O(number of variables in LHS) time */
    else /* var is a new non-operational variable*/
        add var into the list of non-operational variables
        return the operational variables that determines all the variables in the conditions
        /* The computation requires O((number of variables in LHS) x (total
            number of non-operational variables)) time */
}
```

Figure 5.31: The algorithm for computing the exposed variables given a variable.

<ss> is determined by <g> and <s>, because different combinations of the values of these variables in the LHS create different values for <ss>, and <sg> is determined by <g> (as given in the box above Rule-2). For each non-operational variable, the current implementation maintains a set of operational variables that uniquely determine the values of the variable. The boxes in the figure represent this information. Whenever the variable is tested or used elsewhere, the set of variables is used instead. For example, the exposed variables for <ss> in the Rule-3 action are also <g> and <s>.

Figure 5.31 shows the algorithm that computes the exposed variables for a variable, as explained above. The algorithm is executed by a higher level procedure that visits the rules in such an order that for a given non-operational object, the rules that creates the non-operational object are visited earlier than the rules that test the objects in their conditions.

In the algorithm, the system first checks if the given variable is one of the variables that are already known as non-operational. When the variable is one of the non-operational variables, the system returns the set of operational variables pointed by the non-operational variable. Second, if the variable is in the LHS, the algorithm simply returns the variable. Finally, if the variable is a new non-operational variable, add the variable into the list of non-operational variables, and find the set of the operational variables that can be used instead of the non-operational variable. To find the set of operational variables, the system examines each variable in the conditions. While examining each variable in the conditions, the variable is checked whether it is non-operational. If the variable is one of the non-operational variables, the system add the pointed the operational variables into the result. Otherwise, the variable itself is added to the result.

5.4.5 Optimizing the nonlinear structure

Whenever conditions of a nonlinear structure are built, a set of optimizations can be applied before they are compiled into Rete. These optimizations reduce redundant computations without damaging the correctness or increasing the cost. First, a *simple condition* (a condition that is not a nonlinear condition) is ignored when it is tested already. That is, if a simple condition is the same as one of the previous conditions (i.e., conditions between the current condition and the root), or it is contained in one of the previous nonlinear conditions, or it is contained in one of the nonlinear conditions in one of the previous nonlinear conditions, ..., then it is ignored. For example, in Figure 5.32-(a), the simple condition (<p1> ^connected-to <p2>) can be dropped in both cases without reducing the correctness. [2]

A nonlinear condition can be ignored when it is tested earlier by one of the previous conditions, or all of its sub-conditions are tested earlier by the previous conditions. For example, in Figure 5.32-(b), the nonlinear condition marked **shared 1** means that a copy of subrule 1 is tested, and it can be ignored without increasing the cost. Also, in Figure 5.32-(c), the nonlinear condition testing subrule 2 can be ignored because all of its conditions

[2] In the figure, each subrule is tagged with a positive integer. For example, the subrule in Figure 5.32-(a) is tagged with 1. This tagging is used just for the convenience of displaying U′-chunks. By tagging the subrules by the numbers (or some names), whenever a subrule is tested multiple times as multiple nonlinear conditions in a U′-chunk, we can refer to them as the number.

```
(<p1> ^connected-to <p2>)
   •
   •
   •
(<p1> ^connected-to <p2>)
```

```
1 {
   (<s> ^current-position <p1>)
   (<p1> ^connected-to <p2>)
  }
     •
     •
     •
(<p1> ^connected-to <p2>)
```

(a) Duplicate simple conditions

```
1{
   (<s> ^current-position <p1>)
   (<p1> ^connected-to <p2>)
 }
 (<p2> ^connected-to <p3>)
     •
     •
     •
shared 1
     •
     •
     •
```

(b) Duplicate nonlinear condition

```
1{
   (<s> ^current-position <p1>)
   (<p1> ^connected-to <p2>)
 }
 (<p2> ^connected-to <p3>)
     •
     •
     •
2{
   (<s> ^current-position <p1>)
   (<p1> ^connected-to <p2>)
   (<p2> ^connected-to <p3>)
 }
```

(c) All sub-conditions of the subrule 2 are aleady tested

Figure 5.32: Duplicate conditions can be ignored.

are tested already. These duplicate conditions reflect the redundant tests in the original problem solving.

There are more ways of simplifying rule conditions, which are not implemented yet. For example, in Figure 5.33-(a), although all of its sub-conditions are already tested, nonlinear condition testing subrule 1 is not ignored, because the sub-conditions are not tested by its previous conditions. (There is no previous condition for the nonlinear condition.) Also, the nonlinear condition testing subrule 2 also cannot be ignored, because not all of its sub-conditions are already tested. However, the conditions can be simplified, as shown in Figure 5.33-(b), without increasing cost or reducing correctness. The simplification from (a) to (b), in Figure 5.33, requires modification (destruction) of the hierarchical structure of the rule conditions. The current simplification algorithm does not allow such modifications of the hierarchical structure of the conditions; it is left as a future work.

```
(goal <g5> ^operator <o4>)
(<o4> ^type evaluation)
(<o4> ^desired-position <d1>)
2 {
  1 {
     (goal <g4> ^operator <o4>)
     (<o4> ^type evaluation)
     (<o4> ^desired-position <d1>)
    }
   (<d1> ^at <l3>)
 }
```

(a)

```
(goal <g5> ^operator <o4>)
(<o4> ^type evaluation)
(<o4> ^desired-position <d1>)
(<d1> ^at <l3>)
```

(b)

Figure 5.33: A case where optimization is not applied in the current implementation.

5.4.6 An example U′-chunk

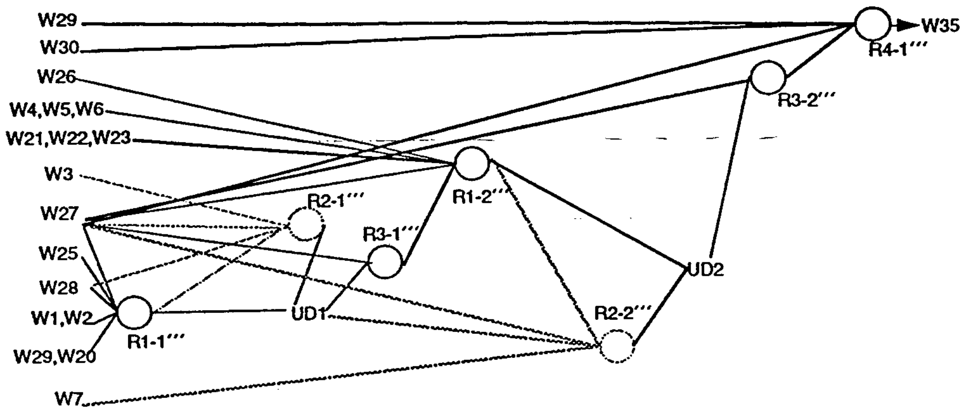
The new U′-chunk built from the I′-chunk in Figure 5.17 is shown in Figure 5.34. The copies of the search-control rules and the subsequent decisions are kept in the structure. The total cost in the number of tokens remains unchanged, instead of increasing.

R4-1‴ in Figure 5.35 shows the hierarchical condition structure of the U′-chunk. There are two unified decisions, and each of them has one decision-sub-condition representing a *best*-preference-created subrule and its consistent acc-subrule. This structure introduces the constraints required to avoid the sources of cost increase, and makes the cost of the learned rule bounded. In general, the number of tokens generated should be the same, or be reduced by applying the above set of optimizations.

Definition 4 (unified-graph) *The unified-graph of a trace-graph is formed by (1) replacing each preference creation from a rule and its use in the connected decision, as a line from the rule to the decision, and (2) replacing each WME creation from a decision and its match in the subsequent rule, by a line from the decision to the rule.*

Definition 5 (unified-trace-subset) *Given the initial WMEs, a U′-chunk A is a unified-trace-subset of a pseudo-chunk B if A's trace-graph is isomorphic to a subgraph of the unified-graph of B's trace-graph.*

Definition 6 (tuple-subset) *Given the initial WMEs, a U′-chunk A is tuple-subset of a pseudo-chunk B if A is a unified-trace-subset of B, and for each rule condition C in A and*



```
(R1-1'''
1 (goal <g> ^state <s>)
1 (<s> ^at <loc1>)
2 (<loc1> ^next <loc2>)
2 (<loc2> ^reachable-by <v1>)
2 (<v1> ^name car)

(R2-1'''
S (goal <g> ^state <s>)
S (<s> ^at <loc1>)
1 (<loc1> ^right <loc2>)
1 (R1-1''')

(UD1
2 (R1-1''')
1 (R2-1''')

(R3-1'''
S (goal <g> ^state <s>)
1 (UD1)

(R1-2'''
S (goal <g> ^state <s>)
1 (R3-1''')
3 (<loc2> ^next <loc3>)
3 (<loc3> ^reachable-by <v2>)
3 (<v2> ^name car)

(R2-2'''
S (goal <g> ^state <s>)
S (R3-1''')
1 (<loc2> ^right <loc3>)
1 (R1-2''')

(UD2
2 (R1-2''')
1 (R2-2''')

(R3-2'''
S (goal <g> ^state <s>)
1 (UD2)

(R4-1'''
S (goal <g> ^state <s>)
1 (<g> ^goal-point <gp>)
1 (<gp> ^at <loc3>)
1 (R3-2''')
-->
1 (<s> ^success <loc3>))
```

Figure 5.34: An interpretation of the U′-chunk that is built while learning a rule from the Grid task. A U′-chunk is created by unifying an I′-chunk.

```
(R4-1‴
S (goal <g> ^state <s>)                                    DS# : decision-sub-condition
 (<g> ^goal-point <gp>)                                    UD#: decision condition
 (<gp> ^at <loc3>)
R3-2‴{
        S (goal <g> ^state <s>)
        UD2{
              DS1{
                    R1-2‴{
                            S (goal <g> ^state <s>)
                            R3-1‴{
                                    S (goal <g> ^state <s>)
                                    UD1{
                                          DS1{
                                                R1-1‴{
                                                        (goal <g> ^state <s>)
                                                        (<s> ^at <loc1>)
                                                        (<loc1> ^next <loc2>)
                                                        (<loc2> ^reachable-by <v1>)
                                                        (<v1> ^name car)
                                                  }
                                                R2-1‴{
                                                        S (goal <g> ^state <s>)
                                                        S (<s> ^at <loc1>)
                                                         (<loc1> ^right <loc2>)
                                                        Shared R1-1‴
                                                  }
                                            }
                                      }
                              }
                            (<loc2> ^next <loc3>)
                            (<loc3> ^reachable-by <v2>)
                            (<v2> ^name car)
                    }
                    R2-2‴{
                            S (goal <g> ^state <s>)
                            Shared R3-1‴
                            (<loc2> ^right <loc3>)
                            Shared R2-1‴
                    }
              }
        }
  }
  ->
  (<s> ^success <loc3>))
```

Figure 5.35: The hierarchical condition structure of the U′-chunk.

its corresponding condition C′ in B, each WME (or tuple, when C is a nonlinear condition) T matching C can be mapped to a unique WME W matching C′ in that T and W contain equivalent information about the variable bindings.

Theorem 5 *Given the initial WMEs, if a U′-chunk A is a tuple-subset of pseudo-chunk B, the number of tokens produced while interpreting pseudo-chunk A is less than or equal to the number of tokens produced by pseudo-chunk B. That is, the number of tokens in A's trace-graph is less than or equal to that in B's trace-graph.*

proof. Because A is a unified-trace-subset of B, each subrule R in A can be mapped to a unique rule, R′ in B. For each condition C in R, since each tuple (or WME) matching the condition can be mapped to a unique WME matching the corresponding condition C′ in R′ (tuple-subset), there will be fewer partial instantiations (tokens) produced while matching R than the tokens produced for R′. Thus, the total number of tokens in A's trace-graph is bounded by the total number of tokens in B's trace-graph.

Theorem 6 *Given the initial WMEs, the number of tokens produced while interpreting a U′-chunk is bounded by the number of tokens of the I′-chunk from which it is created.*

proof. Each subrule in the U′-chunk is created from a unique rule in the I′-chunk. Also, the optimization introduced in subsection 5.4.4 may eliminate some of the subrules, to remove duplicate tests. Thus, the U′-chunk is a unified-trace-subset of the I′-chunk. Since the decision network filters the candidates (as the original decision filters the losers) to produce only the tokens that correspond to the winners, and token compression picks one representative for each set of duplicate instantiations, the U′-chunk is a tuple-subset of the I′-chunk. By theorem 5, the number of tokens produced by the U′-chunk match is bounded by that in the I′-chunk match.

One negative effect of using graph-structured rules is diminished rule readability. The graph structure is rather complex, even with the use of indentation to identify the level of hierarchy. Although not displaying the shared part simplifies the structure a little (Figure 5.36), it is still difficult to understand the structure.

```
(R4-1'''
 (<g> ^goal-point <gp>)
 (<gp> ^at <loc3>)
R3-2'''{
        UD2{
              DS1{
                    R1-2'''{
                            R3-1'''{
                                    UD1{
                                          DS1{
                                                R1-1'''{
                                                         (goal <g> ^state <s>)
                                                         (<s> ^at <loc1>)
                                                         (<loc1> ^next <loc2>)
                                                         (<loc2> ^reachable-by <v1>)
                                                         (<v1> ^name car)
                                                  }
                                                R2-1'''{
                                                         (<loc1> ^right <loc2>)
                                                  }
                                            }
                                      }
                              }
                             (<loc2> ^next <loc3>)
                             (<loc3> ^reachable-by <v2>)
                             (<v2> ^name car)
                     }
                    R2-2'''{
                             (<loc2> ^right <loc3>)
                     }
               }
         }
  }
  -->
  (<s> ^success <loc3>))
```

Figure 5.36: The U′-chunk conditions without the shared sub-parts.

5.5 Summary of Modified Chunking

We have described an application of the proposed modifications to the current chunking process. To incorporate the search control, the new learning algorithm computes the set of preferences that affected the decision. This computation is not essential because we can simply use the set of preferences that participated in the decision. However, it filters excessive search control, and helps produce more general rules. Incorporating search control and keeping the nonlinear structure have required significant modifications to the match algorithm. Not only must the match algorithm be extended to compile the directed-acyclic graph structure, but special decision-sub-nodes for the search-control-created subrules have also been introduced to produce the same control effect without adding exponential overhead. For token compression, different forms of tokens that maintain tuples of the exposed variables' values, instead of tuples of other tokens, are introduced.

Overall, the new learning system requires an implementation of an interpreter for search-control incorporated, token compressing, nonlinear rules.

5.6 Modifying Soar/EBL

Because the primary sources of expensiveness in Soar's learned rules arise in three transformations that are common between chunking and Soar/EBL, the above modifications can also be applied to Soar/EBL. As shown in Figure 5.37, the current sequence of transformations, shown on the left side (original Soar/EBL), has been changed into a new sequence of transformations, shown on the right (new Soar/EBL), by applying the techniques developed for chunking.

The new Soar/EBL shares the same E′-chunk with the modified chunking. To incorporate the search control, the new learning algorithm computes the set of preferences that affected the decision. The next transformation regresses an E′-chunk into a *R′-chunk.* An R′-chunk differs from an R-chunk in that it has additional structures originating from the search-control rules in the E′-chunk. By needing to regress over search control (or decisions), the transformation itself may require extra time, as will be described in Chapter 7. However, the match cost for the R′-chunk does not increase. Finally, an *RU′-chunk*

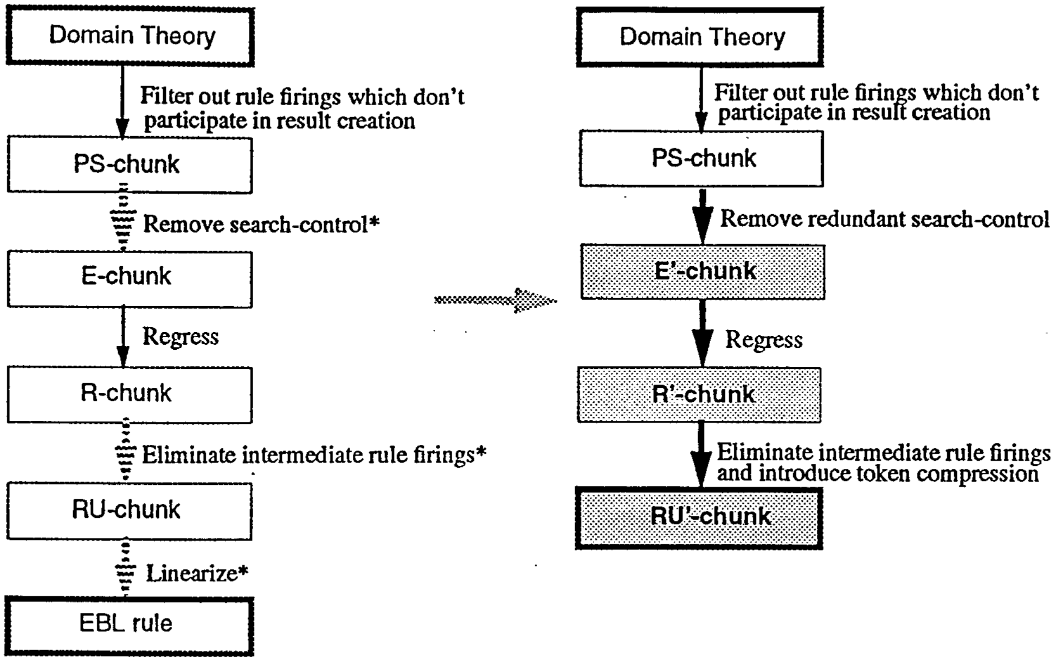

Domain Theory
Filter out rule firings which don't participate in result creation
PS-chunk
Remove search-control*
E-chunk
Regress
R-chunk
Eliminate intermediate rule firings*
RU-chunk
Linearize*
EBL rule
Domain Theory
Filter out rule firings which don't participate in result creation
PS-chunk
Remove redundant search-control
E'-chunk
Regress
R'-chunk
Eliminate intermediate rule firings and introduce token compression
RU'-chunk

Figure 5.37: Modifying Soar/EBL to a new sequence of transformations.

is generated from an R′-chunk, by unifying the separate rules and decisions into a single structure. The token compression optimization is integrated into the last transformation.

The boundedness of the two transformations (E′-chunk ⇒ R′-chunk and R′-chunk ⇒ RU′-chunk) can be proven in the same way as in chunking.

Theorem 7 *The number of tokens produced while interpreting an R′-chunk is bounded by the number of tokens of the E′-chunk from which it is created.*

proof. The R′-chunk has the same set of rules as the E′-chunk because the rules remain the same (trace-subset). The changes made by the transformation either make different variables the same, or constrain the variables as constants. Because of these changes, a rule condition in the R′-chunk matches the same or a subset of WMEs, matching the corresponding condition in the E′-chunk (WME-subset). By theorem 1, the number of tokens produced by the R′-chunk match is bounded by that in the E′-chunk match.

Theorem 8 *Given the initial WMEs, the number of tokens produced while interpreting a RU′-chunk is bounded by the number of tokens of the R′-chunk from which it is created.*

proof. Each subrule in the RU′-chunk is created from a unique rule in the R′-chunk. Also, the optimization introduced in subsection 5.4.4 possibly eliminates some of the subrules to remove duplicate tests. Thus, the RU′-chunk is a unified-trace-subset of the R′-chunk. Since the decision network filters the candidates (as the original decision filters the losers) to produce only the tokens that correspond to the winners, and token compression picks one representative for each set of duplicate instantiations, the RU′-chunk is a tuple-subset of the R′-chunk. By theorem 5, the number of tokens produced by the RU′-chunk match is bounded by that in the R′-chunk match.

Chapter 6

Experimental Results

The purpose of this chapter is to (1) examine if the patterns of cost increase, due to chunking and Soar/EBL, match the earlier analyses presented in Chapter 3 and Chapter 4; and (2) evaluate the modified learning systems that are implemented based on the design details described in Chapter 5.

The first section compares the results from original chunking and Soar/EBL, and the results from different combination of modifications. The domain theory, intermediate pseudo-chunks, and the learned rule are compared in terms of the number of tokens produced during the match. The results from various learning algorithms, as produced by different combinations of the modifications, are compared and analyzed for both chunking and Soar/EBL. In order to interpret and compare intermediate pseudo-chunks and the chunks produced by the different sequences of transformations, we have implemented various extensions of the Rete algorithm, including nonlinear Rete, controlled nonlinear Rete, and controlled nonlinear Rete with token compression. The second section examines the actual problem solving time with the modified chunking and the modified Soar/EBL. The third section discusses the effect of different task representations on the cost of the learned rules. Finally, the last section summarizes the results. The results shown here are all from Soar6 (version 6.0.4), a C-based release of Soar [34] on a Sun SPARCstation-20 with processor 61.

6.1 Match Cost of Different Learning Algorithms

In order to confirm the analyses provided in Chapter 3 and Chapter 4 with experimental results, we have implemented a set of learning algorithms that correspond to the set of initial subsequences of the overall transformation sequence; that is, each learning algorithm starts with the domain theory and generates a distinct type of (pseudo-)chunk. We have also implemented the necessary extensions to the Rete algorithm, which allow all of the types of pseudo-chunks to match and fire. For example, to match a PS-chunk, we have developed a way of closing off internal rule conditions in the PS-chunk from the WMEs generated outside of the PS-chunk. No other WMEs, except for those created by the linked actions, are matched to the conditions of the rules. Also, U-chunks and RU-chunks require the ability to perform a nonlinear match. At each stage, from the domain theory to chunks (or EBL rules), match cost is evaluated by counting the number of tokens required during the match to generate the result and time.

The resulting experimental system has been applied to a variant of the Grid-task; to magnify the effect of each transformation, the task assumes tight connections among the points in a 4×4 grid, as shown in Figure 6.1. The task searches for a path of length four. The results from the task are shown in figure 6.2. The patterns of cost increase match the expectations generated from the earlier analyses of chunking and Soar/EBL, in that a transformation led to increased cost on this task, if and only if it was identified by the analyses as a cost increasing transformation. In both systems, the three transformations — removing search control, unifying, and linearizing — increase the cost. Given the same initial WMEs as in the original problem solving, the numbers of tokens in both systems are the same for all pseudo-chunk pairs. However, Soar/EBL creates a more general rule, as shown in Figure 6.2-(b), because of the one difference between the two transformational sequences. The extra constraint introduced by the variablization step makes the chunk less applicable than the EBL rule. (It will be only applicable when <l3> and <l2> are reachable by the same transportation.) However, it is cheaper to match in different situations. For example, when there is different transportation to reach <l3> and <l2>, as well as the same transportation, the chunk is cheaper to match than the EBL rule.

To examine how the patterns of cost increase change by applying the proposed modifications (either a subset of them or the full set), similar experimental systems are built for different learning algorithms that implement a subset (or the full set) of the modifications. Figure 6.3 shows the match costs of the pseudo-chunks created via different versions of chunking. At each stage from the domain theory to chunks (or pseudo-chunks), the match cost is evaluated by counting the number of tokens required during the match to generate the result. The first column (marked *chunking*) shows the results from the original

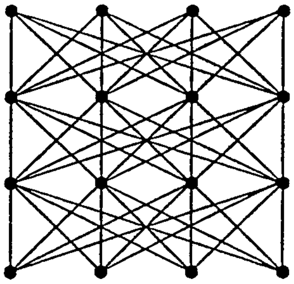


Figure 6.1: A tight grid.

	Number of tokens	
	Chunking	Soar/EBL
Problem Solving	52	52
PS-chunk	42	42
E-chunk	108	108
R-chunk (I-chunk)	108	108
RU-chunk(U-chunk)	198	198
EBL rule (chunk)	215	215

(a) The costs of the pseudo-chunks

```
(chunk                          (EBL rule
(goal <g4> ^desired <d1>)       (goal <g4> ^desired <d1>)
(<d1> ^at <l1>)                 (<d1> ^at <l1>)
(<g4> ^operator <q1> +)         (<g4> ^operator <o2> +)
(<q1> ^to <l3>)                 (<o2> ^to <l3>)
(<l3> ^to <l2>)                 (<l3> ^to <l2>)
(<l2> ^to <l1>)                 (<l2> ^to <l1>)
(<l3> ^reachable-by <t1>)       (<l3> ^reachable-by <t2>)
(<l2> ^reachable-by <t1>)       (<l2> ^reachable-by <t1>)
-->                             -->
(<g4> ^operator <q1> >))        (<g4> ^operator <o2> >))
```

(b) Rules produced by chunking and Soar/EBL

Figure 6.2: Results from a grid task.

SC: incorporate search control
NL: keep the problem solving structure
TC: apply token compression

Grid Task	Number of tokens					
	chunking	+SC	+NL	+SC+NL	+NL+TC	+SC+NL +TC
Domain Theory	52	52	52	52	52	52
PS-chunk	42	42	42	42	42	42
E-chunk(E'-chunk)	108	42	108	42	108	42
I-chunk (I'-chunk)	108	42	108	42	108	42
U-chunk (U'-chunk)	198	83	198	83	102	**44**
chunk	215	72				

44 = 42 + 7 - 8 + 3
7: match cost for decision network
8: additional sharing by dropping architectural local conditions
3: extra match cost of overgeneral rules by dropping architectural local conditions

Figure 6.3: The cost (number of tokens) of various (pseudo-)chunks in chunking.

chunking process. Here, we contrast these results with the ones from other learning algorithms that apply either a subset or the full set of the three optimizations (incorporating search control, keeping the problem-solving structure, and applying token compression) described in Chapter 5. The table shows the results from all feasible combinations of the optimizations. Because token compression is only applicable to nonlinear rules, some of the combinations are unacceptable. For example, token compression alone cannot be implemented.

The second and the third columns show the results from applying one optimization alone: incorporating search control or keeping problem-solving structure. The fourth and fifth columns show the results from applying two modifications: incorporating search control and keeping problem-solving structure, or keeping problem-solving structure and applying token compression. The last column shows the results from the complete combination of the three modifications. Except for the complete combination, other combinations of modifications did not achieve cost boundedness; that is, the number of tokens that are produced while matching the final product is greater than the the number of tokens produced by the problem solving episode.

The increase in the match cost — from 42 to 44 — from the I′-chunk to the U′-chunk (in the last column) arises from two sources: (1) converting the decision process into match effort, and (2) ignoring intractable activities in Soar. Since the decisions are interpreted as additional Rete nodes in the controlled nonlinear Rete, the cost of the decision procedure

SC: Incorporate search control
NL: keep the problem solving structure
TC: apply token compression

Grid Task	Number of tokens					
	Soar/EBL	+SC	+NL	+SC+NL	+NL+TC	+SC+NL +TC
Domain Theory	52	52	52	52	52	52
PS-chunk	42	42	42	42	42	42
E-chunk(E′-chunk)	108	42	108	42	108	42
R-chunk (R′-chunk)	108	42	108	42	108	42
RU-chunk (RU′-chunk)	198	83	198	83	128	52
EBL rule	215	72				

55= 42 + 8 - 1 + 6
8: match cost for decision network
6: extra match cost of overgeneral rules by dropping architectural local conditions
1: ignored condition by dropping architectural local conditions

Figure 6.4: The cost (number of tokens) for various (pseudo-)chunks in Soar/EBL.

(originally performed by the Soar architecture) is converted into a sub-part of the match cost (7 tokens in this case). As explained in Section 5.4, because the increase in the number of tokens has the same complexity as the original decision procedure, the actual total cost does not increase.

The second cause of the cost increase originates from ignoring intractable activities in the problem solving. For example, some of the architectural activities and non-operational negated conditions are ignored in learning. Although this yields an overgenerality of the learned rule, the learning systems do not capture the activities because of their intractability, as described in Section 4.1. In this example, the cost actually decreases from these activities ($-8 + 3 = -5$). Ignoring the architectural part increases the sharing of the conditions in this case, and the cost decreases instead of increasing.

Figure 6.4 shows the match costs of the pseudo-chunks created in different versions of Soar/EBL. The results are from the same Grid task employed for the results shown in Figure 6.3. As in the chunking case, except for the complete combination, the other combinations of modifications could not achieve cost boundedness. They show a similar pattern; the number of tokens that are produced while matching the final product is greater than the the number of tokens produced by the problem solving episode. The increase in the number of tokens from the I′-chunk to the U′-chunk arises from the same sources as in chunking.

Grid task (6) CPU Time (sec)	Learn off	Original	Modified (+SC+NL+TC)
EBL	1.87	15.06	0.84
chunking		1.96	0.76

(a) Results from Grid tasks of path length six

Grid task (7) CPU Time (sec)	Learn off	Original	Modified (+SC+NL+TC)
EBL	2.71	220.34	1.09
chunking		24.61	1.16

(b) Results from Grid tasks of path length seven

Figure 6.5: Average CPU time for Grid tasks.

6.2 Problem Solving Time with the Modified Learning Algorithms

The previous subsection showed results from a single Grid task, applying different combinations of optimizations to chunking and Soar/EBL. In the results, only the complete combination could provide the boundedness. To examine the actual problem solving time with the complete combination, this subsection provides the average problem solving cost (in CPU time) from a set of Grid tasks, instead of one task. The results are from the full set of modifications of both chunking and Soar/EBL.

For experimental efficiency, the results presented assume a 10×10 bounded (but normal) grid. The Grid tasks are searches for paths of length six and paths of length seven. We compared the CPU times from five different versions: without learning, with the rules learned by original Soar/EBL, with the rules learned by original chunking, with the rules learned by modified Soar/EBL, and with the rules learned by the modified chunking. Figure 6.5-(a) shows the average CPU time per problem (in seconds), for a sequence of seven different problems in the Grid task of path length six. Also, Figure 6.5-(b) shows the the average CPU time per problem (in seconds), for a sequence of eight different problems in the Grid task of path length seven. In the path six tasks, the average CPU time from Soar/EBL (15.06) is more than seven times greater than the average CPU time of the system without learning (1.87). In the path seven tasks, the time with Soar/EBL (220.34) is almost eighty times greater than the time without learning (2.71). In the path six tasks, the

```
[   1] (goal <g2> ^problem-space <p9>)
[   1] (<p9> ^name path)
[   4] (<g2> ^operator <x1> +)
[   4] (<x1> ^name goto-loc)
[   4] (<g2> ^state <s17>)
[   4] (<s17> ^at <d1>)
[   4] (<x1> ^at <d1>)
[   4] (<x1> ^to <d2>)
[  16] (<d2> ^conn { <> <d2> <d3> })
[  64] (<d3> ^conn { <> <d3> <d5> })
[ 255] (<d5> ^conn { <> <d5> <d6> })
[1011] (<d6> ^conn { <> <d6> <d7> })
[3989] (<d7> ^conn { <> <d7> <d4> })
[3989] (<g2> ^desired <d1>)
[ 225] (<d1> ^at <d4>)
[ 225] (<s17> ^last-loc <v1>)
```

(a) EBL rule

```
{{{{ [ 4] (goal <g2> ^operator <o7> +)}
{ [ 4] (goal <g2> ^operator <o7> +)
[ 4] (<o7> ^name goto-loc)
[ 4] (<g2> ^problem-space <p22>)
[ 4] (<p22> ^name path)
[ 4] (<o7> ^at <d1>)
[ 4] (<g2> ^state <s18>)
[ 4] (<s18> ^at <d1>)
[ 4] (<s18> ^last-loc <d2>)
[ 1] (<o7> ^to <d2>) }
[ 4] (goal <g2> ^operator <o6> +)}}}
[ 3] (<o6> ^at <d3>)
[ 3] (<d3> ^down <d4>) [ 1] (<o6> ^to <d4>) :
{ [ 4] (goal <g2> ^operator <x3> +)}
[ 3] (<x3> ^at <d5>)
[ 3] (<d5> ^right <d6>) [ 1] (<x3> ^to <d6>)}}
{ { [ 4] (goal <g2> ^operator <o9> +) }
[ 3] (<o9> ^at <d7>)
[ 3] (<d7> ^up <d8>) [ 1] (<o9> ^to <d8>)}}}
[ 1] (goal <g2> ^operator <x3> +)
[ 1] (goal <g2> ^desired <d1>)
{{{{ [ 1] (goal <g2> ^problem-space <p22>)
[ 1] (goal <g2> ^state <s18>)
[ 1] (<x3> ^at <d9>)
[ 1] (<s18> ^at <d9>)
[ 1] (<s18> ^last-loc <v1>)
[ 1] (<x3> ^to <d10>)}
[ 1] (<d10> ^conn { <> <d10> <d9> })}}
{  [ 4] (<d10> ^conn { <> <d10> <d11> })}}}
{ [1] (<d10> ^down <d11>)}
{[ 4] (<d10> ^conn { <> <d10> <d12> })}
{ [1] (<d10> ^right <d12>) }}}
{ [4] (<d10> ^conn { <> <d10> <d13> })
[ 1] (<d10> ^up <d13>)  [ 1] }}
{{{ [ 1] (<d12> ^conn { <> <d12> <d10> }) }
[ 4] (<d12> ^conn { <> <d12> <d14>})
[ 1] (<d12> ^down <d14>)
[ 4] (<d12> ^conn { <> <d12> <d15> })
[ 1] (<d12> ^right <d15>)
[ 4] (<d12> ^conn { <> <d12> <d16> })
[ 1] (<d12> ^up <d16>)  } }  [ 1] (<d1> ^at <d17>)
{ { [  4] (<d15> ^conn { <> <d15> <d17> })}}}}
```

(b) Modified EBL rule

Figure 6.6: Number of tokens of a learned rule in a Grid task.

average CPU time from chunking (1.96) is slightly greater than the time without learning (1.87). In the path seven tasks, the time from chunking (24.61) is more than nine times greater than the time without learning. Chunking and Soar/EBL slow down the problem solving in both cases. (They are expensive-chunk tasks.) Also, the slowdown factors for EBL and chunking in the path seven tasks (81.31 and 9.08) are greater than those in the path six tasks (8.05 and 1.05). However, the time from modified EBL and modified chunking is always less than the time before learning. In the modified learning systems, the time is about half of that without learning. Note that because chunking introduces extra constraints in the rule conditions, it produces less general, but cheaper rules than EBL in some cases.

To examine how match works differently for the rules from the original EBL and those from the modified EBL, we have compared the number of tokens created for two learned rules, one from original EBL and the other from the modified EBL. Figure 6.6 shows the number of tokens at each condition for the match of a learned rule in a Grid task. In the EBL-rule case (Figure 6.6-(a)), there are huge combinations, with a maximum number of 3989 tokens, between the conditions. In the modified-EBL-rule case, as shown in Figure 6.6-(b), the number does not grow to more than 4. In Figure 6.6-(b), braces mark the beginning and ending of subrules in the controlled nonlinear match. This hierarchical structure reflects the problem-solving structure. Shared subrules are not shown in the figure for brevity. The shared conditions across the different sub-parts reflect the multiple usage of those conditions in the original problem solving. This multiple usage keeps the cost bounded, by constraining the sub-parts as they were in the problem solving. Although the rule conditions built by the modified Soar/EBL look rather complex and are difficult to read, they introduce the constraints required to avoid the sources of cost increase and make the cost of the learned rule cheap.

In addition to the Grid task, we applied the system to the 2-Queen, 3-Queen and 4-Queen tasks, which are also known as expensive-chunk tasks. The 2-Queen task places two queens in a three by three grid, without being attacked by each other, as shown in Figure 6.7-(a). The 3-Queen task and 4-Queen task place three and four queens in a four by four grid, as shown in Figure 6.7-(b). The 3-Queen task places only three queens on the grid, and the 4-Queen task places all four queens. Figure 6.8 shows the average CPU time for solving the three tasks. In the 2-Queen task, the times from original EBL and chunking (0.22 and 0.24) are almost the same as the time without learning (0.23). However, the

		Q1
Q2		

(a) 2-Queen task

	Q1		
			Q2
Q3			
		Q4	

(b) 3-Queen task and 4-Queen task

Figure 6.7: Queen task.

2-Queen task CPU Time (sec)	Learn off	Original	Modified (+SC+NL+TC)
EBL	0.23	0.22	0.08
chunking		0.24	0.09

3-Queen task CPU Time (sec)	Learn off	Original	Modified (+SC+NL+TC)
EBL	0.71	9.02	0.16
chunking		10.54	0.15

4-Queen task CPU Time (sec)	Learn off	Original	Modified (+SC+NL+TC)
EBL	1.00	**	0.28
chunking		**	0.25

Figure 6.8: Average CPU time for Queen tasks.

8	3	4
1	5	9
6	7	2

(a)

```
(sp operator-place-tile
   (goal <g> ^problem-space <p>
          ^state <s>)
   (<p> ^name magic)
   (<s> ^square <sq>)
   (<sq> ^number 0 ^name <sq-name>)
   -->
   (<o> ^name place-tile
         ^square-name <sq-name>)
   (<g> ^operator <o>))
```

(b)

Figure 6.9: Magic Square task.

times from the modified EBL and chunking (0.08 and 0.09) are less than half of the time without learning.

In the 3-Queen task, the times from EBL and chunking (9.02 and 10.54) are more than ten times greater than the time without learning (0.71). As in the Grid task cases, the slowdown factor increases as the size of the task increases. The modified learning algorithms provide boundedness as in the above tasks. The problem solving with modified EBL and chunking produces the same results more than four times faster than the problem solving without learning.

In the 4-Queen task, the system could not even finish learning with both original EBL and chunking. The number of tokens for the learned rule became over eight million and the system could not allocate enough memory. Still, the CPU times from modified EBL and chunking are bounded by the time without learning. The time without learning is greater than the time from modified EBL and chunking by factors of 3.57 and 4, respectively.

We also applied the system to the Magic Square task[61], another known expensive-chunk task. This task involves placing tiles 1 through 9 in empty squares of 3×3 grid one at a time. If the sum of horizontal, vertical, and diagonal lines are different with the current tile placement, the task fails. Otherwise, the process can continue placing tiles until it fills all nine squares. The results show the same pattern as the results from the Queen tasks. With original EBL and chunking, the system could not finish learning. However, the CPU times with modified EBL and chunking (3.47 and 1.86) are bounded by the time without learning (6.91). The time without learning is greater than the time with modified EBL and chunking by factors of 1.99 and 3.72, respectively.

The bar charts shown in Figure 6.11 summarize the results from the above tasks.

Magic Square task CPU Time (sec)	Learn off	Original	Modified (+SC+NL+TC
EBL	6.91	**	3.47
chunking		**	1.86

Figure 6.10: Average CPU time for the Magic Square task.

6.3 Effects of Different Task Representations

When a task is given, there is usually more than one way to represent it. This section examines the effect of different representations on the cost of learned rules. Figure 6.12-(a) shows the costs of the learned rules (the final products), from one of the length six Grid tasks. The results are produced from a Grid task, in which the evaluation order is based on better preferences among the directions (right > left > up > down). We can represent the same task in a different way. For example, we can evaluate the operator to go to the right first by employing a best preference (right > others). Its results are shown in Figure 6.12-(b).

In the two tables, the first column shows the cost of problem solving without learning. In both chunking and EBL, the cost without learning is the same. The second column shows the cost of the learned rules in standard Soar/EBL and chunking. The third and fourth columns show the costs of the learned rules with only one modification: incorporating search control, or keeping problem-solving structure. The fifth and sixth columns show the results from applying two modifications: incorporating search control and keeping problem-solving structure, or keeping problem-solving structure and token compression. The last column shows the results from the complete combination.

In Figure 6.12-(a), for both original EBL and chunking, the cost of matching the learned rules is expensive without the modifications. (The given task is one of the expensive-chunk tasks.) Keeping the problem-solving structure alone, or applying token compression to the nonlinear structure without search control, produces very expensive rules, and the system cannot finish the task (because of memory exhaustion). With the complete combination (last column), the time after learning (1.15 and 1.12) is less than the time before learning (2.00). In this case, incorporating search control, with or without the problem-solving structure, produces cheaper rules than the original chunking or Soar/EBL.

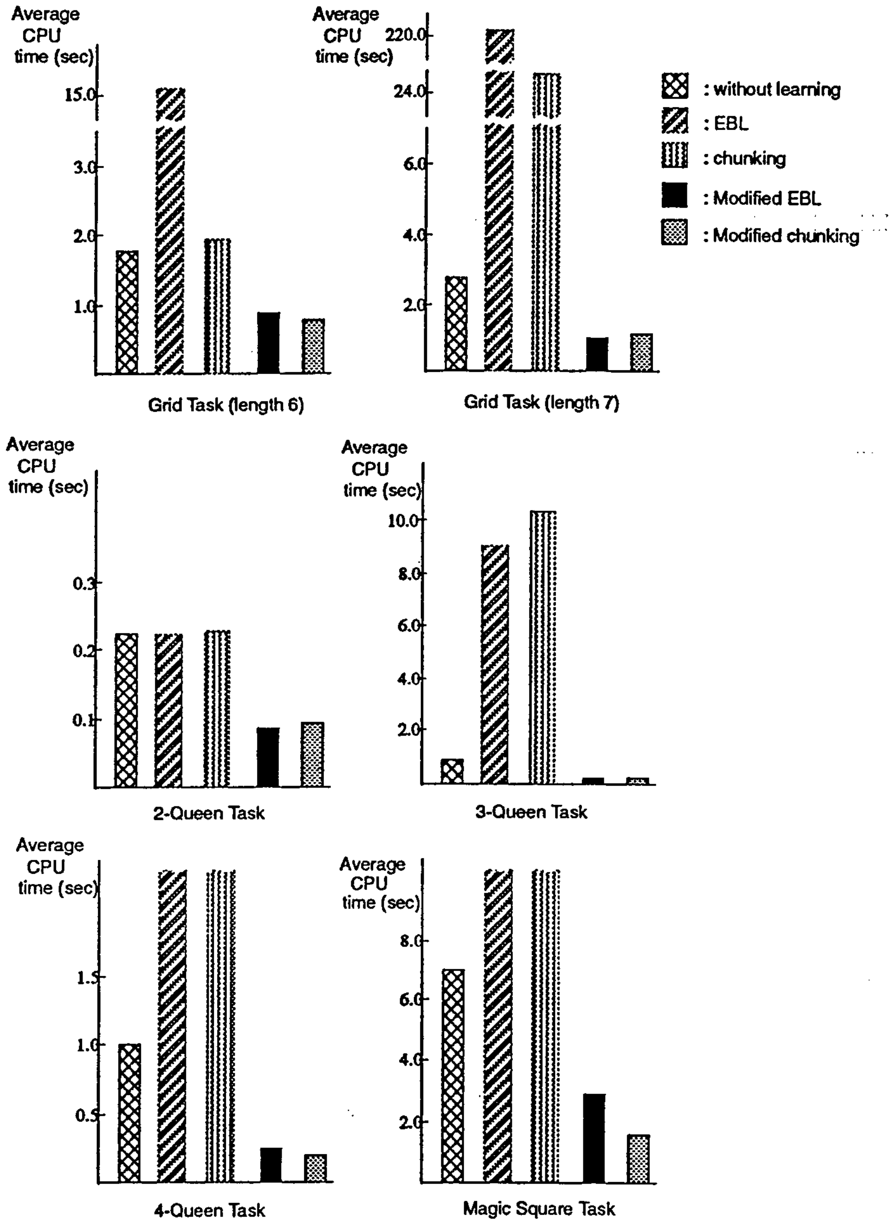


Figure 6.11: Summary of the results.

CPU Time (sec)	Learn off	Original	+SC	+NL	+SC+NL	+NL+TC	+SC+NL +TC
EBL	2.00	23.31	1.96	**	1.46	**	**1.15**
chunking		2.89	**0.44**	**	0.88	**	**1.12**

(a) Results from a length six Grid task

CPU Time (sec)	Learn off	Original	+SC	+NL	+SC+NL	+NL+TC	+SC+NL +TC
EBL	0.98	0.26	**	3.25	0.30	2.55	**0.29**
chunking		0.25	45.70	2.01	0.31	1.86	**0.28**

(b) Results from a different representation of the Grid task

Figure 6.12: Results from different representations of a Grid task.

The most efficient rules are produced by incorporating search control alone in chunking; the cost is 0.44 second.

With the different representation, the results show a different pattern (Figure 6.12-(b)); chunking with search control does not produce the best result. In fact, it creates expensive rules; the cost is 45.70 second in this case, while the cost without learning is only 0.98 second. Also, note that the original chunking and Soar/EBL provide better results in this case (0.26 and 0.25) than the learning systems with the complete combination (0.29 and 0.28). Although the complete combinations could not give the best results, they still provide boundedness. The U′-chunk (and the RU′-chunk) solves the same problem three times faster than the original problem solving.

6.4 Summary and Discussion

The above results have demonstrated that the modifications of the learning algorithm based on the implementation details discussed in Chapter 5 actually bound the cost after learning to the cost before learning (except for cost introduced by overgeneralization), at least for the domains investigated. The original learning algorithms and the algorithms that implement a subset of the three modifications can produce better results than the complete combination in some cases. In other cases, however, the same algorithms can produce expensive rules. The complete combination, though it does not produce the best results in all cases, consistently provides boundedness.

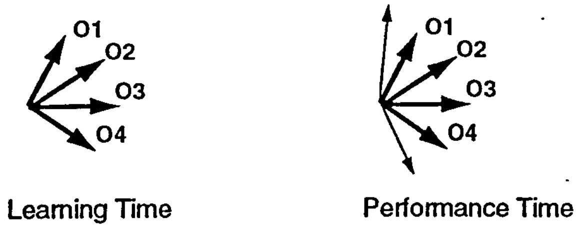


Figure 6.13: Options can increase at performance time.

One positive side-effect of incorporating search control is that it removes one possible source of overgeneralization in Soar. Although search control is not supposed to affect the correctness of results generated in problem spaces, it sometimes does. In situations in which results are returned from a problem space before the *goal test* (i.e., the test of whether the desired goal is achieved) succeeds, or where the goal test is itself overgeneral, the search control may affect the correctness of the result. Under such circumstances, not including this search control in learning can yield overgeneral learned rules. By including the search control in the explanation of the result, the modified chunking and modified Soar/EBL remove this source of overgenerality.

There is an issue that arises because of the option taken to interpret the decision procedure in the U′-chunk. It only occurs when there are more options available at performance time than at learning time. In particular, if the conditions learned to discriminate among the options available at learning time are not sufficient to discriminate among these new options, an additional match search may be introduced. For example, when there were a fixed number of operators during learning, such as the operators moving toward the four directions in the Grid task, the learned rules may not be suitable for problems that have more operators available, as shown in Figure 6.13. This is related to the *masking effect* [64], where learned knowledge masks original-problem solving knowledge; thus the system can produce low quality solutions. This did not occur in the previous experiments, but it could happen in other domains.

This problem can be fixed by employing additional constraints in the decision network. Whenever the number of winners is not equal to one, either by failing to find a winner or by having more than one winner, the decision network can stop further matches, considering the decision as failed.

Chapter 7

The effect of the New Learning Algorithms

The set of modifications of chunking and Soar/EBL introduces different computational complexities into the learning algorithms. For example, in order to incorporate the search-control trace in learning, the system has to perform extra computations for the additional conditions introduced by search-control rules. Other modifications by the subsequent transformations combined with this modification may also require different computational complexities. This chapter examines the overheads in the new sequence of transformations. First we describe the costs of original chunking and Soar/EBL, and then the costs of the modified learning algorithms are compared with them. Finally, we examine the effect of the modified learning systems on larger scale tasks, based on the cost analyses.

7.1 Cost of Chunking

As explained in subsection 2.2.1, chunking consists of four steps: (1) filtering out traces that do not participate in result creation, (2) removing search control and building a backtrace, (3) variablizing, and (4) unifying the structure and creating a new rule.

The first and the second step are performed by the backtracing process; while it filters the unnecessary rule firings, it also removes the search control by only examining the task-definition traces (traces of the task-definition rules). Because the backtracing process examines all the task-definition traces linked to the result until it reaches the supergoal (operational) elements, its complexity depends on the number of instantiated conditions of all the task-definition traces linked to the result. Thus, the cost of the first two steps is

$O(C)$, where C is the total number of instantiated conditions in the task-definition traces that are linked to the result.

The third step variablizes the identifiers by examining each item in the instantiated conditions. Its complexity depends on the total number of items in the instantiated conditions, including duplicate instances of them. In Soar, the items in an instantiated condition are *id*, *attribute*, and *value*; so, the number of items in an instantiated conditions is constant. Thus, the complexity of this step is $O(C)$.

The last step builds a new rule based on the variablized conditions. In the process of building rule conditions, the set of operational conditions — conditions created from the supergoal elements — are reordered by a heuristic algorithm to improve the match performance. The reordering algorithm (adapted from [8] and Soar version 6.0.4) is shown in Figure 7.1. The algorithm uses a backtrack-free heuristic to reduce the complexity. It examines the set of conditions one at a time, and tries to pick cheapest condition in terms of the estimated match cost of having the condition be the next condition. The complexity of the algorithm is $O(o^3)$, where o is the number of operational conditions. (There are three nested loops in case there is a tie for minimum cost.)

Based on the cost of each step, the total complexity of chunking is $O(C + C + o^3) = O(C + o^3)$.

7.2 Cost of Soar/EBL

Soar/EBL is similar to chunking except for the way of determining the variable names in the learned rule. As described in Chapter 4, Soar/EBL performs the following steps: (1) filtering out traces that do not participate in the result creation, (2) removing search control and building the explanation, (3) constructing the explanation structure, (4) regressing, and (5) creating a new rule. (1) and (2) are the same as in chunking. Also, (5) corresponds to the last step in chunking.

The explanation structure is built by replacing the instantiations with the rules. The variable names are replaced with unique names so that there are no common variables across the rules. To make the variable names unique, the system examines all the variable instances in the LHS (condition part) of all the participating rules, and it requires O(total number of variable instances in the LHS of all participating rules). Because the number of variable instances in a condition is approximately constant in Soar, and the total number

```
reorder (conditions) {
  bound_vars ⇐ root variables
       /* "root variables" are the variables used for the id field of conditions
        that start with "goal" or "impasse". Usually, this is just the variable "<g>" */
  REPEAT until there are no more remaining conditions:
  eligible_conds ⇐ remaining conditions whose id field is in bound_vars
  for each c in eligible_conds, find_cost(c, bound_vars)
    if one has the minimum cost, output that condition
    else there's a tie for minimum cost, so do a one-step lookahead:
       for each tied minimal-cost condition tied_c:
          temp_bound_vars ⇐ bound_vars plus any variables used in tied_c
          temp_eligibles ⇐ remaining conditions (except tied_c) whose id field
                  is in temp_bound_vars
          for each temp_c in temp_eligibles, find_cost (temp_c, temp_bounded_vars)
          find the minimum of these cost(temp_c)'s
       output the condition tied_c whose minimum cost(temp_c) is smallest
          (if there's still a tie, just pick the first one)
    add the variables in the picked condition into the bound_vars
}
find_cost(c, bound_vars) /* an estimate of the match cost of having c be the next condition */
    /* look at the id, attribute, value fields of c and check which ones have either constants
       or variables in bound_vars */ {
 If id field is unbound, return 10000
 If attr field is unbound but value field is bound, return 8
 If value field is unbound but attr field is bound, return 8
 If all three fields are bound, return 1.
}
```

Figure 7.1: The reordering algorithm in Soar.

of conditions in the explanation structure is equal to the total number of instantiated conditions in the backtrace, the complexity of building the explanation structure is $O(C)$.

The current implementation of EBL regression builds a *substitution list* based on action and condition pairs that collide in the explanation structure, and applies the substitutions in the substitution list to the variables. The number of collisions (of actions and conditions) depends on the number of decisions in the problem solving. For each decision, the system adds new pairs (based on *id*, *attribute*, and *value* fields of the action and the condition) after making sure that these pairs are not already in the substitution list. The complexity of this computation is $O(\text{(number of decisions)} \times \text{(length of the substitution list)})$. Because the length of the substitution list is $O(\text{total number of variables})$, and $O(\text{total number of variables}) \leq O(C)$, the cost of building the substitution list is $O(C \times D)$, where D is the total number of decisions.

Finally, the algorithm unifies the variables by applying the substitution list, which requires $O(\text{(total number of variable instances)} \times \text{(length of the substitution list)})$. Thus, the complexity of applying the substitution list is $O(C^2)$. Based on the cost of each step, the total complexity of Soar/EBL is $O(C + C + C \times D + C^2 + o^3) = O(C^2 + C \times D + o^3)$.

If we use different representations for substitutions, such as a hash table or extra pointers, the cost can be reduced. Building the substitutions can be $O(D)$ instead of $O(C \times D)$ because checking if a pair is not already in the substitutions takes a constant time. Also, the complexity of applying the substitutions can be reduced to $O(C)$. In this case, the total cost becomes $O(C + D + o^3)$.

7.3 Cost of Performing the New Chunking Algorithm

7.3.1 Cost of Domain Theory⇒PS′-chunk

As in chunking (or Soar/EBL), elimination of unnecessary rule firings can be performed by following the rule traces linked to the result. Because the set of traces in new chunking includes search-control traces as well as task-definition traces, the cost of following the traces is $O(S)$, where S is the total number of instantiated conditions in all the traces (including search-control traces) that are linked to the result.

7.3.2 Cost of PS′-chunk⇒E′-chunk

As described in Section 5.2, this transformation computes a set of relevant preferences capturing the full decision context (instead of the full set of participating preferences) for each decision. The preference collection algorithm has been presented in Figure 5.8. The computational complexity of the algorithm is $O((\text{number of candidates in a decision})^4)$ per decision. Thus, the total complexity is $O(D \times c^4)$, where c is the maximum number of candidates in a decision. As described in Section 5.2, if the system preprocesses *better/worse* preferences or employs additional indices, the cost can be reduced to $O(D\times c^3)$. Based on this computation, the system builds a data structure, called a *Decision* for each decision. A *Decision* keeps the set of participating preferences for its decision. Also, these *Decisions* are connected by a linked list called the *Decision_list.* Each *Decision* in the *Decision_list* is represented as a letter D in Figure 5.16. These *Decisions* are transformed by the subsequent transformations (as the rules in pseudo-chunks are transformed), and used in building decision conditions as shown in Figure 5.25.

7.3.3 Cost of E′-chunk⇒I′-chunk

This step performs the variablization (constraining variables by instantiation) for the rule conditions in an E′-chunk. Because an E′-chunk maintains search-control rules as well as task-definition rules, the system requires extra computation for processing the conditions in the search-control rules, while the transformation from an E-chunk to an I-chunk variablizes only the conditions of the task-definition rules. The total complexity of this transformation is $O(\text{total number of variable instances in the rule conditions in the E}'\text{-chunk})$. Because the number of variables in a simple condition (a condition that is not a nonlinear condition) is approximately constant, the complexity is $O(\text{total number of simple conditions in the E}'\text{-chunk})$. (From now on in this chapter, a *condition* means a simple condition.) Because the total number of conditions in an E′-chunk is bounded by the total number of instantiated conditions in the explanation, the cost of the variablization is $O(S)$.

7.3.4 Cost of I'-chunk⇒U'-chunk

As described in Section 5.4, this transformation unifies separate rules and decisions into one structure, and applies token compression.

The exposed variable computing algorithm for token compression is presented in Section 5.4, and the algorithm is shown in Figure 5.30. In the algorithm, the system first checks if the given variable is one of the variables that are already known as non-operational. To check if the variable is a member in the list, the algorithm examines items in the list one by one. Thus, the complexity of this check is O(number of non-operational variables). Second, if the variable is in the LHS, the algorithm simply returns the variable. The computation of verifying whether or not the variable is in the LHS requires O(total number of variable instances in LHS) times for each variable in the action. Finally, if the variable is a new non-operational variable, add the variable into the list of non-operational variables, and find the set of the operational variables that can be used instead of the non-operational variable. To find the set of operational variables, the system examines each variable in the conditions. While examining each variable in the conditions, the variable is checked if it is non-operational, by scanning the non-operational variable list. The complexity of the computation is O((total number of variable instances in the conditions) × (total number of non-operational variables)).

The overall complexity of computing the exposed variables for both operational and non-operational variables in the actions is O((total number of variable instances in actions) × (total number of non-operational variables) × (maximum number of variable instances in the LHS of a rule)).

By employing different data structures and spending more memory space, the complexity can be reduced. If the system uses a hash table or extra pointers for the non-operational variables, the complexity of checking if a variable is non-operational is O(1). Also, when the variable is a new non-operational variable, finding the set of operational variables that can be used instead of the non-operational variable takes just O(total number of variable instances in the conditions). Thus, the total cost of computing the exposed variables is O((total number of variable instances in actions) × (maximum number of variable instances in the LHS of a rule)).

O(total number of variable instances in action) $<$ O(total number of variable instances in all conditions). Also, O(total number of variable instances in all conditions) $=$ O(total number of conditions in all participating rules) ≤ O(total number of instantiated conditions in the traces). Thus, O(total number of variables in the action) < O(S). In the same way, O(total number of non-operational variables) < O(S), and O(maximum number of variables in the LHS of a rule) < O(S). Thus, the complexity of the algorithm is at most $O(S^3)$. With different data structures, as described above, the cost can be $O(S^2)$.

To build U′-chunk conditions based on the graph structure of the problem solving, the system traverses the rule firing structure. Each operational condition becomes a new condition, and each non-operational condition becomes a new nonlinear condition after the system traverses its subrule. Thus, the complexity of building new rule conditions depends on the total number of conditions of all the rules (search-control rules and task-definition rules) participated in the I′-chunk.

While building conditions, each *Decision* becomes a decision condition. The current implementation organizes the subrules that participated in the decision into the structure shown in Figure 5.25, in order to build a decision condition. In the structure, each non-acceptable preference is paired with the *acceptable* preference which has the same value field with the preference. The preferences in a *Decision* are grouped by their preference types, so that there is a linked list for each preference type. To find the consistent *acceptable* preference given a non-acceptable preference, the system examines each item in the linked list of *acceptable* preferences, and it requires O(number of candidates) times. Because each non-acceptable preference (computed by the preference collection algorithm) can filter at least one candidates the number of non-acceptable preferences in a *Decision* is bounded by the number of candidates. Thus, the cost of building a decision condition is $O(c^2)$ time where c is the maximum number of candidates in a decision. If the system maintains pointers from each preference to the *acceptable* preference that has the same value field, the complexity becomes just constant. Thus, the cost can be O(c).

Application of the optimization described in subsection 5.4.4 requires more computation. For each newly built nonlinear condition, the system checks whether all of its conditions is already tested by earlier conditions. This requires the complexity of O((number of conditions in a nonlinear condition) × (number of conditions already tested)) for building each nonlinear condition. Therefore, the complexity of the optimization is O((total number of nonlinear conditions) × (maximum number of conditions in a nonlinear condition) × (total number of conditions)). O(total number of nonlinear conditions) < O(total number of conditions) and O(maximum number of conditions in a nonlinear condition) <

O(total number of conditions). Thus, the complexity of the optimization is $O(S^3)$. If the system employs a hash table to check if a condition are already tested, its complexity can be O((number of conditions in a nonlinear condition) × (maximum number of conditions in a nonlinear condition)), and it is $O(S^2)$.

The total overhead of this transformation is $O(S^3 + D \times c^2 + S^3) = O(S^3 + D \times c^2)$. With a different data structure, as described above, it can be $O(S^2 + D \times c)$.

7.3.5 Total overhead of the new chunking

The total overhead of the new learning algorithm is $O(D \times c^4 + S + S^3 + D \times c^2)$. Because the second term is bounded by the third term, and the fourth term is bounded by the first term, the total overhead is $O(D \times c^4 + S^3)$. If the system preprocesses preferences and employs additional data structures, it will consume more memory space, but in return the total overhead reduces to $O(D \times c^3 + S^2)$.

7.4 Cost of Performing the New EBL Algorithm

The computational complexity of the transformations in the modified EBL is similar to that of the modified chunking, except for the regression transformation (E′-chunk⇒R′-chunk). As in Soar/EBL, the transformation consists of three steps: (1) making the variable names unique across the rules, (2) building *substitutions* based on action and condition pairs that collide in the explanation structure, and (3) applying the *substitutions* to the variables. First, to make the variable names unique across the rules, the current implementation examines all the variable instances in the LHS of the rules, and it requires O(total number of variable instances in the LHS of the rules) times. Because the total number of variable instances in the LHS of the rules in an E′-chunk is bounded by the number of instantiated conditions in the backtrace, the complexity is $O(S)$.

As in Soar/EBL, the substitution list is built based on action and condition pairs that collide in the explanation structure. The number of collisions (of actions and conditions) depends on the number of decisions (either trivial or non-trivial). For each trivial decision, we can just add new pairs (based on *id*, *attribute*, and *value* fields of the action and the condition) after making sure that these pairs are not already in the substitution list. The complexity of this computation is O((number of trivial decisions) × (length of the substitution list)).

In non-trivial decision cases, non-winners (filtered candidates) cannot be unified with the connected condition, because they have different values. To unify the variables in the search-control rules (as well as those in the task-definition rules), the system adds more pairs to the substitution list. For each filtered candidate, the system finds the rule action that proposed the candidate (by an *acceptable* preference), and also the action that filtered the candidate by creating a preference against it. Then, the system adds new pairs based on the *id*, *attribute*, and *value* fields of both the *acceptable*-preference-created action and the preference-created action. That is, for each non-acceptable preference, the system finds the *acceptable* preference that proposed the same candidate. (Each preference maintains a pointer to the action that created the preference.) This requires O((number of candidates)^2 × (length of substitution list)) time for each decision in the current implementation. Thus, computing the substitution list needs $O(D \times c^2 \times$(length of the substitution list)) time. Because the length of the substitution list is O(total number of variables), and O(total number of variables) = O(total number of conditions), the complexity is $O(D \times S \times c^2)$.

Finally, the algorithm applies the substitutions in the substitution list to the variables, which requires O((total number of variables) × (length of the substitution list)) time. Thus the complexity of applying the substitutions is $O(S^2)$. Based on the cost of each sub-step, The total complexity of the regression is $O(S + S \times D \times c^2 + S^2) = O(S \times D \times c^2 + S^2)$.

If the system employs additional data structures (additional pointers or a hash table) for substitutions, checking if a pair is not already in the substitutions needs only a constant time. Also, if the system maintains extra pointers from each preference to the *acceptable* preference that has the same value field, finding the rule action that proposed the candidate needs a constant time. Thus, the complexity of the regression reduces to $O(S + D \times c + S) = O(S + D \times c)$.

Based on the cost of each transformation in the new chunking algorithm in the above section, if we add the cost of the other transformations, the total overhead of the new EBL algorithm is $O(D \times c^4 + S^3 + S \times D \times c^2)$. With different data structures, it can be $O(D \times c^3 + S^2)$.

Grid Task	total cost	
	Chunking	Soar/EBL
Original	$O(C + o^3)$	$O(C^2 + C \times D + o^3)$
New	$O(D \times c^4 + S^3)$	$O(D \times c^4 + S^3 + S \times D \times c^2)$
New + Hash tables	$O(D \times c^3 + S^2)$	$O(D \times c^3 + S^2)$

Table 7.1: Total overhead of different learning algorithms.

7.5 Summary of the Overhead Analyses

Table 7.1 summarizes the costs of the different learning algorithms. The two new learning algorithms (new chunking and new Soar/EBL) require more time than the original algorithms to process search control ($O(D \times c^4)$) and to unify conditions ($O(S^3)$). If the system employs additional data structures, such as hash tables or extra pointers, it will consume more memory space, but in return, the cost can be reduced by an order of magnitude (from S^3 to S^2, and from c^4 to c^3).

7.6 Effects on Larger Scale Tasks

This section examines the generality of the modified learning systems to larger scale tasks. We examine the cases of (1) learning a large number of chunks, (2) a large number of conditions in the rules, (3) a large number of rule firings in the problem solving, and (4) a large number of candidates per decision.

1. Learning a large number of chunks (average-growth effect): As the number of chunks grows, the cost of using (matching) chunks usually increases. Recent work on this problem in linear Rete [9] has shown that the application of additional optimization (Rete/UL) greatly reduces the average-growth effect. In some tasks, it has been possible to learn over one million rules while still allowing their efficient use. Because our new algorithm assumes nonlinear Rete instead of linear Rete, to be able to generalize the results, the effect of Rete's optimizations (including state saving, sharing, and Rete/UL) on nonlinear Rete should be examined.

   Nonlinear Rete can provide the same state saving as linear Rete. It can preserve the previous matches in alpha memories and beta memories, though the structure of the tokens is a little different, as described in Section 3.4. Also, nonlinear Rete provides a sharing optimization, which can share nonlinear conditions as well as linear conditions. As in the case of linear Rete, sharing is possible for the same initial conditions among different rules. In addition to that, the same nonlinear conditions within a rule (by being tested multiple times in the hierarchical structure), or across different rules can share the same nodes in the network. For example, for U′-chunk shown in Figure 5.35, R1-1‴ and the nonlinear conditions marked as "Shared R1-1‴" use the same nodes in the nonlinear-Rete network for the match. Actually, all the conditions marked as "S" or "Shared" share tokens with some other conditions of the rule. Also, when different rules have the same nonlinear conditions, they can share the same network. Detailed computational analysis of the effect of this sharing needs to be performed.

Rete/UL introduces the elimination of unnecessary processing in the *join* nodes, which maintains constant time per token for learning a large number of rules. Although the current nonlinear Rete implementation does not employ this optimization, we expect that similar optimizations can be implemented in nonlinear Rete. The effect of this extension also needs to be analyzed.

2. A large number of conditions in the rules: As described in the prior sections, the overhead of the new chunking and the new Soar/EBL is $O(D \times c^4 + S^3)$ and $O(D \times c^4 + S^3 + S \times D \times c^2)$, respectively, where S is the total number of instantiated conditions in the explanation, D is the number of decisions, and c is the maximum number of candidates in a decision. Because the learning time is a cubic factor of the number of conditions, increasing the number of conditions can affect the cost of learning time by a polynomial factor. This increase can be reduced by introducing different data structures. Currently, variables and preferences in the decisions are maintained as linked lists. By changing them into a hash table structure, or adding direct pointers among the structures, the cost can be reduced to $O(D \times c^3 + S^2)$ for both chunking and Soar/EBL by an order of magnitude.

3. A large number of rule firings in the problem solving: The complexity of the new learning algorithm depends on the number of rule conditions, the number of decisions that participated in the problem solving, and the number of candidates per decision. Because the first two numbers increase as the number of rule firings grows, a large number of rule firings can affect the learning time by a polynomial factor.

As mentioned above, this increase can be reduced by changing the data structure (and spending more memory space).

4. A large number of candidates per decision: As described above, the costs of the new chunking and the new Soar/EBL depend on the number of candidates per decision to the power of four. As described in the second item above, the cost of both the new chunking and the new Soar/EBL can be reduced to $O(D \times c^3 + S^2)$. Also, if there are no *better/worse* preferences, or the learning skips the preference collection algorithm (employing all the preferences participated in the decisions), the cost becomes $O(D \times c^2 + S^2)$ or $O(D \times c + S^2)$. However, in the worst case, as the number of candidates per decision increases, the learning time can still grow fast, by a square or a linear factor.

Chapter 8

Related Work

This chapter describes work related to solving the utility problem and performing the transformational analysis. Section 7.1 examines approaches taken to solve the utility problem. Section 7.2 describes other transformational analyses of learning.

8.1 Solving the Utility Problem

The goal of our research is to *provide a relative solution to the utility problem without restricting the expressiveness of the learned knowledge.* In the preceding chapters, we discussed the *transformational approach,* which consists of two steps : (1) finding the complete set of sources that can make learned rules expensive, and then (2) modifying the learning process to avoid these sources. The transformational approach can provide a relative solution to the utility problem in that it ensures the cost of using learned rules is bounded by the cost of problem solving. Also, it itself does not impose any restriction on the expressiveness or completeness (finding all solutions) to achieve such boundedness. In this section, we present alternative approaches to solve the utility problem, and discuss how they are less suited to achieve our goal.

Figure 8.1 illustrates the structure of the following discussion. The structure subdivides the approaches taken to solve the utility problem, based on the types of learning algorithms the approaches are addressing and the issues on which they are focusing. The sequence of marked boxes emphasizes the part this research concentrates on. We address the utility problem for learning search-control rules by EBL, and focus on achieving a relative bound.

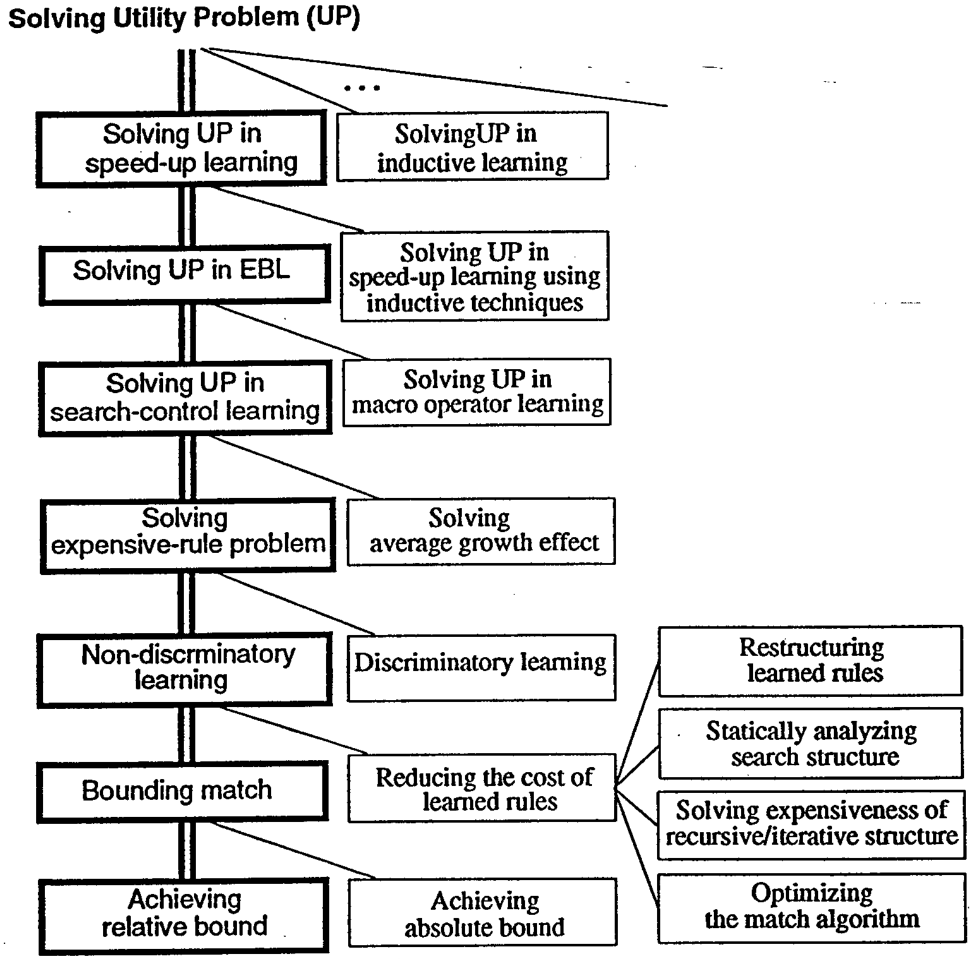

Solving Utility Problem (UP)
...
Solving UP in speed-up learning
SolvingUP in inductive learning
Solving UP in EBL
Solving UP in speed-up learning using inductive techniques
Solving UP in search-control learning
Solving UP in macro operator learning
Solving expensive-rule problem
Solving average growth effect
Non-discrminatory learning
Discriminatory learning
Restructuring learned rules
Statically analyzing search structure
Bounding match
Reducing the cost of learned rules
Solving expensiveness of recursive/iterative structure
Achieving relative bound
Achieving absolute bound
Optimizing the match algorithm

Figure 8.1: Related issues in solving the utility problem.

8.1.1 Speed-up learning vs. inductive learning

Since the utility problem has been identified in speed-up learning systems, the term "utility problem" has also come to be used for a variety of other issues and phenomena. For example, the term has been used for the accuracy and completeness of inductive learning systems, in which the learning is intended to achieve a better classification, but not necessarily speed-up [24, 48, 25, 5].

There are various inductive methods, including set-covering approaches, and splitting approaches. Set-covering approaches, including AQ [40], construct disjunctive-normal-form expressions from training examples. These approaches suffer from an inaccuracy problem as the number of disjuncts increases. In order to produce more accurate (but less complete) hypotheses, Michalski[40] applied truncation techniques. Also, splitting approaches, including ID3 [51], recursively split the set of training data by choosing an appropriate feature or feature value pair. ID3 suffers from an overfitting problem as the decision tree becomes deeper. To alleviate this problem, pruning techniques have been developed [52].

Our research focuses on the utility problem for speed-up learning. However, if the transformational analysis technique is applicable to the inductive learning systems, it would be interesting to study how an analysis of accuracy changes and completeness changes through an inductive learning system; and it would also be interesting to examine whether the analysis can be used as a tool for revealing the sources of incompleteness and inaccuracy. The approaches taken in the earlier work have focused on either how to simplify the hypothesis, or when to stop learning based on the performance evaluation, instead of finding the sources of inaccuracy and incompleteness in the learning algorithm.

On the other hand, applying inductive techniques to chunking or Soar/EBL would be useful to estimate and correct the overgenerality of the chunks, caused by architectural activities or local negated conditions.

8.1.2 EBL vs. speed-up learning using inductive techniques

There are speed-up learning methods using inductive techniques. For example, an *evaluation function*, as a real-valued function, can be learned to estimate how close a given state is to the goal [22, 33, 55]. Also, the *strength of operators*, as quantities associated with the operators, can be learned to indicate how successful the operators have been in the past

[18]. State evaluation functions can be used to guide the search, such as best-first search. Also, given the strength of operators, the problem solver can choose operators in an order of decreasing strength, or choose them probabilistically according to their strength.

Our work focuses on EBL, the most widely used speed-up technique [41, 7, 14, 27, 23], rather than on the above techniques. Once we solve the utility problem in EBL, the results may help guide similar analyses of other speed-up learning techniques.

8.1.3 Search-control learning vs. macro-operator learning

EBL can be used for acquiring either *search-control knowledge* or *macro-operators*. Search-control knowledge can be learned in the form of search-control rules, or through other kinds of control information, including an evaluation function represented as a set of EBL rules. A macro-operator is formed by aggregating a sequence of primitive operators into one operator. This new operator can be used with the primitive operators, and takes a "big step" in the problem solving.

The addition of macro-operators to the original operators increases the branching factor of a problem solver's search, and the extra search performed by the macro-operators can be redundant with the search performed by the original operators, as described in [42, 13]. Learning search-control knowledge obviates this problem, since it learns to control the problem solving activity instead of changing the search space by adding new operators. However, it still suffers from the utility problem, because the cost of matching the control knowledge can be expensive. This research investigates the utility problem for control knowledge learning, and does not address the problem of branching factor increase or search redundancy for macro operators.

8.1.4 Expensive-chunk problem vs. average-growth effect

Research on the utility problem in EBL has raised two key issues: (1) the cost of individual rule (the expensive-rule problem) and (2) the cost of interactions among the rules, or the effect of learned rules on problems other than the ones for which the rules were learned (the average-growth effect).

Our research focuses on the expensive-rule problem. The average-growth effect is the effect of chunks on the problems the chunks cannot solve, and it depends on the amount of match effort performed for the chunks that do not fire. There are favorable properties in Soar and Rete that protect the system from performing the unnecessary match effort. Soar's problem solving is formulated as a hierarchy of modular problem spaces. By testing the problem space in the beginning of the rule match, the system can save the match effort for a chunk when it is irrelevant to the current problem. Also, even when a chunk that does not fire addresses the same problem space as the problem to be solved, Rete's sharing for the same test pattern across the rules can obviate redundant tests. As shown by Doorenbos[9], the average-growth effect can be further reduced by adding more optimizations that prevent unnecessary match effort that does not affect the results. These optimizations and other solutions to the average-growth effect must be combined with our solutions, but it is a topic for future work.

8.1.5 Non-discriminatory learning vs. discriminatory learning

One class of approaches to the expensive-rule problem is discriminatory learning, where the utility of learned knowledge is evaluated and only the useful knowledge is kept. Several systems assume a fixed distribution of problems, and select those performance-system transformations that allow increased utility. These originate in Minton's [41] utility evaluation, where PRODIGY/EBL measures the utility in terms of savings and cost of a rule, and rules are deactivated if their utility is estimated as negative. Greiner and Jurisica [19] proposed an algorithm called PALO that navigates through the space of performance elements. PALO selects a new performance element, that is strictly better than the current performance element until it reaches a local optimum. The utility analysis in PALO computes expected performance based on the test cases from a fixed distribution. This is similar to the approach taken in Composer [16], which adds a control rule to the system only if it shows incremental utility. The utility is determined by the expected (problem solving) cost for a sequence of problems. The information filtering model [39] proposes a more general framework for discriminatory learning, and defines various methods for eliminating harmful knowledge from the learning system. Discrimination processes, called filters, may be inserted to remove such knowledge. The filters include selective experience, selective attention, selective acquisition, selective retention, and selective utilization.

These approaches are useful for filtering out low utility rules when the learning system produces such rules. They are also beneficial for removing obsolete or harmful knowledge in the system, if it exits. However, these discriminatory learning approaches need to evaluate the utility of the candidate rules, and these evaluations may become a part of the utility problem. For example, given a set of *n* interacting transformations, if the goal is to find the optimal subset, one must consider all 2^n subsets. Although they often use various techniques, such as hill-climbing, based on assumptions about the problems, utility evaluation is a complex problem itself. Also, gathering reasonable utility data is a difficult problem, as described in [17]. Thus, these evaluation processes may change the problem of high cost rules into the problem of high cost learning. The transformational approach is different from the above discriminatory approaches. The objective of the transformational approach is learning cheap rules from the beginning, instead of choosing high utility rules during or after learning. As described in Chapter 7, the overhead of making sure the learned rule is cheap is polynomial in three numbers — the total number of decisions in the problem solving, the maximum number of candidates for a decision, and the total number of instantiated conditions in the rule traces — by the maximum power of four. The maximum power can be reduced to two or one, as explained in Chapter 7.

8.1.6 Providing a bound vs. reducing the cost

The goal of our work is to ensure that the cost of using learned rules is bounded by the cost of problem solving without the learned rules. There has been a lot of work on reducing the cost of learned rules, without guaranteeing such boundedness. Some approaches have restructured the learned rules to semantically equivalent ones in order to reduce the match cost of the rules. Partial evaluation in PROLEARN [50] simplifies the learned rules by exploiting domain constraints. COMPRESSOR [41] in PRODIGY simplifies rules or combines multiple rules to generate less expensive descriptions. It employs domain knowledge, partial evaluation, reordering, and logical equivalences to find a better structure. Although these restructuring approaches are useful for producing cheaper rules, they are incomplete in the sense that they do not guarantee that all of the sources of expensiveness to be extracted. Also, if the transformation process is complex, it may suffer from the problem of high cost learning. The objective of this research is to learn cheap enough rules so that the system does not have to restructure them afterwards.

Some other approaches have preserved the search structure in the learned knowledge. In Shell and Carbonell's work [59], they employ iterative constructs in learned macro-operators to capture the iterative paths found during the problem-space search. These iterative macro-operators are then used in a way that guarantees that they take the same path followed in the problem space. Shavlik [58], and Subramanian and Feldman [60] learn recursive and iterative concepts by generalizing the explanation structure. These approaches are close to 'incorporating search control in learning' — one of the optimizations we applied in the new chunking algorithm — in that the search paths are reflected in the learned rules. However, these approaches do not completely solve the expensive-rule problem, because not all expensive chunks arise from losing search structures. For example, losing efficiencies (such as sharing) stemming from the hierarchical explanation structure cannot be captured in these approaches. The transformational approach is more general than these approaches because it captures the factors that determined the entire problem-space search, rather than limited search structures; thus, it can handle all of the causes of expensive chunks.

There is another class of approaches which statically analyze search structure or problem space structure, and produce cheap rules even before problem solving. Taylor and Korf [66] developed a technique to detect duplicate operator sequences from a small breadth-first search in order to control the redundancy in problem solving. STATIC [12] employs a depth-first search in the graph structure of goal/subgoal and operator relationships, and extracts control rules from the non-recursive subgraphs it finds. These approaches can provide useful information by preprocessing the search space. However, these approaches do not utilize the dynamic aspect of problem solving, losing where to focus in the structure, and thus have potential disadvantages with respect to EBL [49]. Although DYNAMIC [49] has provided an intermediate solution by introducing problem distribution sensitivity into STATIC, these approaches still handle only limited structures in problem-space search. They focus on operator/goal relationships, and do not address other aspects of problem solving, including the optimizations employed in the problem solving, such as sharing.

The match cost of learned rules can be reduced by employing better match algorithms. Rete and Treat[44] are currently the best known rule match algorithms. Also, there has been a lot of work to improve these algorithms[26, 2, 38, 21, 9, 6], which show even more improvement in the match performance. Although they themselves cannot provide a relative solution for learning, adapting these techniques to our learning system would be useful for improving the absolute performance of the problem solving. Although our system is already built on an optimized Rete, we can employ other optimizations whenever they are applicable.

8.1.7 Relative solution vs. absolute solution

The transformational approach can provide a relative solution to the utility problem in that it ensures the cost of using learned rules is bounded by the cost of problem solving. Also, the approach does not impose any restriction on expressiveness or completeness to achieve such boundedness. An *absolute solution* is defined as one that provides a guaranteed bound on the match of the learned rules, regardless of the original problem-solving cost. However, the absolute solutions presented so far all require either a set of restrictions on expressiveness or impose incompleteness.

One approach that provides an absolute solution is the *unique-attribute restriction* [61]. The unique-attribute restriction disallows object attributes from having more than one value. While getting a bound on match, the unique attribute restriction has several drawbacks. Not only does it reduce the expressibility of the system, it also reduces the generality of the rules. It sometimes requires a large number of rules for the same knowledge which could be expressed by a single rule [61]. In our work, the incorporation of search control in modified chunking and Soar/EBL can also reduce the generality of the learned rules. However, the results [29] show that the unique-attribute version learns more specialized rules than simply incorporating search control.

There is another approach, called the *instantiationless tree* approach [65], that also provides a guaranteed bound on match. The approach restricts the rule system by eliminating rule instantiations and disallowing some equality tests. By losing equality tests, it also reduces the generality of the learned rules. We have not yet performed the comparison between the rules from our modified learning systems and the rules from the instantiationless tree. However, we expect that instantiationless tree will generate rules that have similar generality as those from the original chunking or Soar/EBL, but its tasks will suffer from the restriction on the equality tests in learning new rules.

Some other approaches sacrifice completeness to provide predictability of the response time. For example, Haley[20] sets a bound to a number of parameters in the Rete algorithm, including the number of tokens, to limit the total match cost. Also, Barachini and Verteneul[3] provide upper bounds by limiting the values of attributes to be in certain interval. However, not only these approaches impose incompleteness, they also can perform (bounded) exponential search.

8.2 Other Transformational Analyses of Learning

The following approaches are similar to our work in their use of a transformational analysis to speed-up the problem solving.

Segre and Elkan [57] analyzed EBL as a structural application of three explanation-transformation operators: specialize, generalize, and prune. Also, the work extends the algorithm by introducing two more operators to provide higher utility rules. In their work, the term *transformation* means each operation applied to the EBL explanation to produce the learned rule. Thus, a sequence of transformations produces EBL rules from the explanation as in our transformational analysis. Although their transformations are used as a tool for describing EBL algorithms, as in our work, they are not used as a tool for measuring the costs of intermediate products. Their focus is on how to combine the transformation operators, guided by control heuristics, to produce a better EBL algorithm.

Another approach, by Bostrom [4], applies three transformation operators (definition, unfolding, and folding) to transform the domain theory into a more efficient form. His transformations are different from ours in that their sequence of transformations changes the domain theory into another, using the transformation operators, instead of changing the problem solving episode into a new rule.

Keller and Mostow's [28, 47] transformations are close in spirit to our transformations in that a sequence of transformations reformulates non-operational knowledge into operational knowledge. However, the meaning of *transformation* in the transformational analysis is different from the meaning of the term used for their work. The transformations here describe the changes to the input knowledge according to the given learning algorithm (such as EBL). However, transformations in operationalization are guided by some control knowledge heuristics, not directly related with any learning algorithm. The focus of their transformations is simply performance improvement (by operationalizing the given knowledge) rather than using the transformations as a tool for analyzing the given learning algorithm.

All of the transformational analyses presented above are also different from our analysis in that the focus of the approaches and their resulting algorithm development was on speedup rather than on boundedness, and on STRIPS-type macro-operator learning, rather than on search control learning.

Chapter 9

Conclusion

This chapter summarizes important results from the thesis, and presents issues for future work.

9.1 Summary

Many learning systems suffer from the utility problem; time after learning is greater than time before learning. Discovering how to assure that learned knowledge will in fact speed up system performance has been a focus of research in explanation-based learning (EBL). This thesis focused on *ensuring that the cost of using learned rules is no more than the cost of problem solving*. In order to achieve this goal, we proposed the *transformational approach*, which consists of two steps: (1) finding the complete set of sources that can make learned rules expensive, and then (2) modifying the learning process to avoid these sources. Also, to find the set of sources of expensiveness, this research introduced a novel way of analyzing the learning process — the *transformational analysis*. The essence of the analysis is *to decompose the learning process into a sequence of transformations in which the cost of intermediate products can be computed.* By computing and comparing the match cost of each intermediate product, the cost changes through the learning were measured and isolated within the steps where the transformations occur.

The thesis uses chunking and Soar/EBL as a vehicle for the investigation. Also, the match algorithm employed for this research is a state-of-the-art Rete algorithm. Chunking has been decomposed into a sequence of transformations from a problem solving episode to the matching and firing of a chunk. The match cost of each intermediate product (pseudo-chunk) was measured by counting the number of tokens produced in the match to generate the result. By analyzing the transformations, we identified a set of sources which can make the output chunk expensive. In addition to identifying the sources, the analysis also pointed the way towards modifications of the transformational sequence that could potentially eliminate the sources. The set of sources and the proposed modifications are:

1. *Removing search control ⇒ incorporate search control in learning.* By incorporating search control in the explanation structure, the match process for the learned rule can focus on the path that was actually followed.

2. *Losing efficiencies stemming from the problem-solving structures ⇒ keep the problem-solving structure.* By keeping the graph structure employed in the problem solving, the efficiencies can be reinstated.

3. *Disrupting the optimizations based on equivalent knowledge ⇒ preprocess knowledge before it is used.* By preprocessing the knowledge either by grouping the equivalent knowledge or by selecting one of them as a representative, an equivalent optimization can be achieved.

To be able to more easily generalize the resulting analysis to other EBL systems, we have implemented a general EBL algorithm in Soar (Soar/EBL) and analyzed its performance. The Soar/EBL process that goes from a problem solving episode to a learned rule has been decomposed into a sequence of transformations. The transformations have been mapped to the corresponding transformations in chunking, and have been compared in terms of cost and generality. This comparison has revealed that the primary sources of expensiveness in Soar's learned rules arise in three transformations that are common between chunking and Soar/EBL. This comparison also has revealed that Soar/EBL yields the same sources of overgenerality as does chunking. The differences between Soar/EBL and chunking has been localized within a single transformation, where chunking overspecializes with respect to Soar/EBL.

The application of the proposed solutions (for both chunking and Soar/EBL) requires significant change in the underlying Soar architecture, especially the match algorithm. The required alterations include the following:

1. *Computing the necessary search-control rules and eliminating redundant rules:*

To be able to incorporate the search control in learning without introducing redundant conditions, an algorithm has been developed to compute and collect the relevant search-control rules at each decision point.

2. *Developing a new match algorithm (controlled nonlinear Rete) which can interpret search-control incorporated rules that maintain the problem-solving structures:*

   Because intermediate preference and WME creations should be converted into subtasks of the match process, the Rete algorithm has been extended to perform such tasks. In this extended Rete, conditions are hierarchically combined via *join* nodes that compare a pair of tokens instead of a token and a WME. This requires the ability to create hierarchically structured tokens; that is, a token must now be a sequence of WMEs or tokens, instead of a sequence of WMEs. Also, to interpret the search control semantics, an extra node type (decision-sub-node) has been introduced.

3. *Introducing token compression in controlled nonlinear Rete:*

   Another new form of token has been introduced. Instead of forming the tokens as tuples of WMEs or tokens, the extended Rete generates tuples of values of the variables which are going to be needed later in the match process.

The above set of modifications has been applied to both chunking and Soar/EBL, and the original sequence of transformations has been converted into a new sequence of transformations. The experimental results for the expensive-chunk tasks imply that the time after learning is consistently less than the time before learning with the modified learning algorithm. The original learning, and the algorithms that implement only subparts of the full modifications, sometimes produce better results than the fully modified learning algorithm. However, the learning algorithm which gives the best results in one problem may produce expensive rules in another, while the full modification always provides boundedness.

In summary, the primary contributions of this thesis include: performing a transformational analysis of the EBL algorithm; identifying the sources of expensiveness; and providing a new algorithm based on the solutions for the sources. We performed such analysis in the context of Soar, and identified the sources of expensiveness, along with the modifications that can eliminate the sources. Also, the alterations required in the learning algorithm and the underlying match algorithm to support the modifications have been designed and implemented. The experimental results indicate that, at least for the domains investigated, the new learning algorithm provides relative boundedness; the run time after learning is consistently less than the run time before learning, except for the changes caused by the architectural axioms.

9.2 Future Work

One negative effect of using graph-structured rules is *diminished rule readability*. Even with the use of indentation to identify the level of hierarchy, the sharing of sub-conditions is still difficult to understand. One way of relieving this problem is by further simplifying the structure of the rules. There are more ways of simplifying the graph structure than the ones already implemented. The approaches include the modification of the nonlinear structure into a more efficient structure, as described in Section 5.4. By implementing those, we may improve the match performance as well as the readability of the rules in the modified learning algorithms.

The new learning algorithms (for both chunking and Soar/EBL) need to be combined with a solution to the average growth effect. The earlier work on the average growth effect in chunking has shown that it is possible to learn large number of rules without hurting overall system performance. However, because the rules created by the new learning algorithms can be different from the rules created by chunking, the problem still needs to be addressed in terms of the new learning algorithm.

As described in Chapter 6, the performance-time effect may be avoided by employing additional constraints in the decision network. The additional constraints needs to be implemented, and the analysis of the effect of unexpected alternatives during performance time is required.

The overgenerality caused by Soar's architectural activities can lead to cost changes. How much cost increase they can generate needs to be analyzed and it should be combined with the analysis done for non-architectural activities.

The results presented in Chapter 6 are based on known expensive-chunk tasks. Experimental results from a wider range of tasks, both other expensive-chunks tasks and non-expensive-chunk tasks, should be performed to generalize our solutions.

Reference List

[1] A. Acharya. Personal communication. 1996.

[2] A. Acharya and M. Tambe. Collection-oriented match : Scaling up the data in production systems. Technical Report CMU-CS-92-218, Computer Science Department, Carnegie-Mellon University, 1992.

[3] F. Barachini and G. Verteneul. The challenge of real-time process control for production systems. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 705–709, 1988.

[4] Henrik Bostrom. Improving example-guided unfolding. In *Proceedings of ECML-93*, pages 124–135, 1993.

[5] B. Carlson, J. Weinberg, and D. Fisher. Search control, utility, and concept induction. In *Proceedings of the Seventh International Conference on Machine Learning*, pages 85–91, 1990.

[6] B. Cho. *Efficient Production Match and CSP Solving*. PhD thesis, University of Southern California, 1996.

[7] R. Dechter. Enhancement schemes for constraint processing: Backjumping, learning, and cutsets decomposition. *Artificial Intelligence*, 41:273–312, 1990.

[8] B. Doorenbos. Personal communication. 1994.

[9] B. Doorenbos. *Production Matching for Large Learning Systems*. PhD thesis, Carnegie-Mellon University, 1995.

[10] B. Doorenbos. Personal communication. 1996.

[11] B. Doorenbos and M. M. Veloso. Knowledge organization and the utility problem. In *Proceedings of the Third International Workshop on Knowledge Compilation and Speedup Learning*, 1993.

[12] O. Etzioni. Why Prodigy/EBL works. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 916–922, 1990.

[13] O. Etzioni. An asymptotic analysis of speedup learning. In *Proceedings of the Ninth International Workshop on Machine Learning*, pages 135–142, 1992.

[14] Y. E. Fattah and P. O'Rorke. Explanation-based learning for diagnosis. *Machine Learning*, 13:35–70, 1993.

[15] C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17–37, 1982.

[16] J. Gratch and G. Dejong. COMPOSER: A probabilistic solution to the utility problem in speed-up learning. In *Proceedings of the Tenth National Conference on Aritificial Intelligence*, pages 235–240, 1992.

[17] J. Gratch and G. DeJong. A statistical approach to adaptive problem solving. *Artificial Intelligence*, 88(1):101–142, 1996.

[18] J. J. Grefenstette. Credit assignment in rule discovery systems. *Machine Learning*, 3:25–45, 1988.

[19] R. Greiner and I. Jurisica. A statistical approach to solving the EBL utility problem. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 241–248, 1992.

[20] P. V. Haley. Real-time for rete. In *Proceedings of ROBEXs'87: The Third Annual Workshop on Robotics and Expert Systems*, 1987.

[21] E. N. Hanson and M. S. Hasan. Gator: An optimized discrimination network for active database rule condition testing. Technical Report TR-93-036, CIS Department, University of Florida, 1993.

[22] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on System Science and Cybernetics*, 4:100–107, 1968.

[23] R. W. Hill and W. L. Johnson. Situated plan attribution. *Artificial Intelligence in Education*, 6(1):35–66, 1995.

[24] L. B. Holder. The general utilty problem in machine learning. In *Proceedings of the Seventh International Conference on Machine Learning*, pages 402–410, 1990.

[25] L. B. Holder. Empirical analysis of the general utility problem in machine learning. In *Proceedings of the Ninth International Workshop on Machine Learning*, pages 249–254, 1992.

[26] T. Ishida. Optimizing rules in production system programs. In *Proceedings of the Seventh National Conference on Artifical Intelligence*, pages 699–704. Morgan Kaufmann, 1988.

[27] S. Katukam and S. Kambhampati. Learning explanation-based search control rules for partial order planning. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 582–587, 1994.

[28] R. M. Keller. Learning by re-expressing concepts for efficient recognition. In *Proceedings of the National Conference on Artificial Intelligence*, pages 182–186, 1983.

[29] J. Kim and P. S. Rosenbloom. Constraining learning with search control. In *Proceedings of the Tenth International Conference on Machine Learning*, pages 174–181, 1993.

[30] J. Kim and P. S. Rosenbloom. Learning efficient rules by maintaining the explanation structure. In *Proceedings of the Thirteenth National conference on Artificial Intelligence*, pages 763–770, 1996.

[31] J. Kim and P.S. Rosenbloom. Mapping explanation-based learning onto Soar: The sequel. Technical Report :Transformation analyses of learning in SOAR. ISI/RR-95-4221, Information Sciences Institute and Computer Science Department University of Southern California, 1995.

[32] J. Kim and P.S. Rosenbloom. Transformation analyses of learning in Soar. Technical Report ISI/RR-95-4221, Information Sciences Institute and Computer Science Department University of Southern California, 1995.

[33] R. E. Korf. Real-time heuristic search. *Artificial Intelligence*, 42:189–211, 1990.

[34] J. E. Laird, C. B. Congdon, E. Altmann, and R. Doorenbos. *Soar User's Manual: Version 6*, 1 edition, 1993.

[35] J. E. Laird, A. Newell, and P. S. Rosenbloom. Soar: An architecture for general intelligence. *Artificial Intelligence*, 33:1–64, 1987.

[36] J. E. Laird, P. S. Rosenbloom, and A. Newell. Chunking in Soar: The anatomy of a general learning mechanism. *Machine Learning*, 1, 1985.

[37] J. E. Laird, P. S. Rosenbloom, and A. Newell. Overgeneralization during knowledge compilation in Soar. In *Proceedings of the Workshop on Knowledge Compilation*, pages 46–57, 1986.

[38] H. S. Lee and M. I. Schor. Match algorithms for generalized Rete networks. *Artificial Intelligence*, 54:249–274, 1992.

[39] S. Markovitch and P. D. Scott. Information filtering : Selection mechanism in learning systems. *Machine Learning*, 10(2):113–151, 1993.

[40] R. S. Michalski. How to learn imprecise concepts: A method based on two-tiered representation and the AQ15 program. III, 1986. In press.

[41] S. Minton. Quantitative results concerning the utility of explanation-based learning. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 564–569, 1988.

[42] S. Minton. Issues in the design of operator composition systems. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 304–312, 1990.

[43] S. Minton. Personal communication. 1993.

[44] D. P. Miranker. Treat: A better match algorithm for AI production systems. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 42–47, 1987.

[45] T. M. Mitchell, R. M. Keller, and S. T. Kedar-Cabelli. Explanation-based generalization – a unifying view. *Machine Learning*, 1(1):47–80, 1986.

[46] R. J. Mooney and S. W. Bennett. A domain independent explanaion-based generalization. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, pages 551–555, 1986.

[47] D. J. Mostow. Machine transformation of advice into a heuristic search procedure. In J. Carbonell R. Michalski and T. Michell, editors, *Machine Learning: An Artificial Intelligence Approach*. Tioga Press, Palo Alto, CA, 1983. In press.

[48] M. Pazzani and D. Kibler. The utility of knowledge in inductive learning. *Machine Learning*, 9(1):57–94, 1992.

[49] M. A. Perez and O. Etzioni. DYNAMIC: A new role for training examples in EBL. In *Proceedings of the Ninth International Workshop in Machine Learning*, pages 367–372, 1992.

[50] A. E. Prieditis and J. Mostow. PROLEARN: Towards a Prolog interpreter that learns. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 494–498, 1987.

[51] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986.

[52] J. R. Quinlan. Simplifying decision trees. *International Journal of Man-Machine Studies*, 27:221–234, 1987.

[53] P. S. Rosenbloom and J. E. Laird. Mapping explanation-based generalization onto Soar. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, pages 561–567, Philadelphia, 1986. AAAI.

[54] P. S. Rosenbloom, J. E. Laird, A. Newell, and R. McCarl. A preliminary analysis of the Soar architecture as a basis for general intelligence. *Artificial Intelligence*, 47(1-3):289–325, 1991.

[55] A. L. Samuel. Some studies in machine learning using the game of checkers. IBM Journal, Vol. 3, No. 3, July, 1959.

[56] D. J. Scales. Efficient matching algorithms for the Soar/Ops5 production system. Technical Report KSL-86-47, Knowledge Systems Laboratory, Department of Computer Science, Stanford University, 1986.

[57] A. Segre and C. Elkan. A high-performance explanation-based learning algorithm. *Artificial Intelligence*, 69:1–50, 1994.

[58] Jude W. Shavlik. Aquiring recursive and iterative concepts with explanation-based learning. *Machine Learning*, 5:39–70, 1990.

[59] P. Shell and J. Carbonell. Empirical and analytical performance of iterative operators. In *The 13th Annual Conference of The Cognitive Science Society*, pages 898–902. Lawrence Erlbaum Associates, 1991.

[60] D. Subramanian and R. Feldman. The utility of EBL in recursive domain theories. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 942–949, 1990.

[61] M. Tambe. *Eliminating combinatorics from production match.* PhD thesis, Carnegie-Mellon University, 1991.

[62] M. Tambe. Personal communication. 1996.

[63] M. Tambe, D. Kalp, A. Gupta, C. L. Forgy, B. G. Milnes, and A. Newell. Soar/PSM-E: Investigating match parallelism in a learning production system. In *Proceedings of the ACM/SIGPLAN Symposium on Parallel Programming: Experience with applications, languages, and systems*, pages 146–160, 1988.

[64] M. Tambe and P. S. Rosenbloom. On the masking effect. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 526–533, 1993.

[65] M. Tambe and P. S. Rosenbloom. Investigating production system representations for non-combinatorial match. *Artificial Intelligence*, 68(1):155–199, 1994.

[66] L. A. Taylor and R. E. Korf. Pruning duplicate node in depth-first search. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 756–761, 1993.

[67] F. Zerr and J. G. Ganascia. Comparison of chunking with EBG implemented onto Soar. Universite Paris-Sud, Orsay, France and Universite Pierre et Marie Curie, Paris, France. October, 1989, Unpublished.